TortoiseGit

A Git client for Windows

Version 2.16.0

Lübbe Onken (TortoiseSVN)
Simon Large (TortoiseSVN)
Frank Li
Sven Strickroth

TortoiseGit: A Git client for Windows: Version 2.16.0

by Lübbe Onken (TortoiseSVN), Simon Large (TortoiseSVN), Frank Li, and Sven Strickroth

Table of Contents

Preface xiv
1. Audience xiv
2. Reading Guide xiv
3. TortoiseGit is free! xiv
4. Community xv
5. Acknowledgments xv
6. Terminology used in this document xvi
1. Introduction 1
1.1. What is TortoiseGit? 1
1.2. TortoiseGit's History 1
1.3. TortoiseGit's Features 1
1.4. Installing TortoiseGit 2
1.4.1. System requirements 2
1.4.2. Installation 3
1.4.3. Language Packs 3
1.4.4. Spell checker 3
2. TortoiseGit Daily Use Guide 5
2.1. Getting Started 5
2.1.1. Icon Overlays 5
2.1.2. Context Menus 5
2.1.3. Drag and Drop 7
2.1.4. Common Shortcuts 8
2.1.5. Authentication 8
2.1.6. Maximizing Windows 9
2.2. Create Repository 9
2.3. Clone Repository 9
2.4. Checking Out A Working Tree (Switch to commit) 10
2.5. Committing Your Changes To The Repository 12
2.5.1. The Commit Dialog 12
2.5.2. Change Lists 14
2.5.3. Excluding Items from the Commit List 14
2.5.4. Commit only parts of files 15
2.5.5. Commit Log Messages 15
2.5.6. Commit Progress 17
2.6. Getting Status Information 18
2.6.1. Icon Overlays 18
2.6.2. Explorer Properties 19
2.6.3. Status 20
2.6.4. Viewing Diffs 22
2.7. Pull and Fetch change 22
2.8. Push 25
2.8.1. Branch 25
2.8.2. Destination 26
2.8.3. Options 26
2.9. Sync 26
2.9.1. Branch 27
2.9.2. Destination 27
2.9.3. Options 27
2.10. Daemon 28
2.11. Browse All Refs 28
2.12. Submodules 30
2.13. Change Lists 32
2.14. Log Dialog 34
2.14.1. Invoking the Revision Log Dialog 34
2.14.2. Revision Log Actions 35

2.14.3. The revision graph ... 35
2.14.4. Commit messages and branch/tag indicators ... 35
2.14.5. Getting Additional Information ... 36
2.14.6. Filtering Log Messages ... 41
2.14.7. Navigation ... 43
2.14.8. Statistical Information ... 44
2.14.9. Refreshing the View ... 46
2.15. Revision Graphs ... 46
2.15.1. Revision Graph Nodes ... 47
2.15.2. Using the Graph ... 48
2.15.3. Refreshing the View ... 48
2.16. Reference Log ... 48
2.17. The Repository Browser ... 49
2.18. Viewing Differences ... 51
2.18.1. File Differences ... 51
2.18.2. Line-end and Whitespace Options ... 52
2.18.3. Comparing Version ... 52
2.18.4. Diffing submodules using Submodule Diff Dialog ... 54
2.18.5. Diffing Images Using TortoiseGitIDiff ... 55
2.18.6. External Diff/Merge Tools ... 57
2.19. Adding New Files ... 57
2.20. Ignoring Files And Directories ... 59
2.20.1. Pattern Matching in Ignore Lists ... 60
2.21. Deleting, Copying, Moving and Renaming Files and Folders ... 61
2.21.1. Deleting files and folders ... 62
2.21.2. Copying Files and Folders ... 62
2.21.3. Moving Files and Folders ... 62
2.21.4. Changing case in a filename ... 63
2.21.5. Dealing with filename case conflicts ... 63
2.21.6. Deleting Unversioned Files ... 64
2.22. Undo Changes ... 64
2.23. Cleanup ... 65
2.24. Reset ... 66
2.25. Stash Changes ... 68
2.26. Bisect ... 69
2.27. Branching/Tagging ... 70
2.27.1. Creating a Branch or Tag ... 70
2.28. Merging ... 72
2.29. Cherry picking ... 74
2.30. Rebase ... 75
2.31. Resolving Conflicts ... 76
2.31.1. Special conflict cases ... 78
2.32. Creating and Applying Patches and Pull Requests ... 79
2.32.1. Creating a Patch Serial ... 79
2.32.2. Sending patches by mail ... 80
2.32.3. Applying a single Patch File ... 81
2.32.4. Applying a Patch Serial ... 82
2.32.5. Creating a pull request ... 83
2.33. Who Changed Which Line? ... 84
2.33.1. Blame for Files ... 84
2.34. Exporting a Git Working Tree ... 86
2.35. Integration with Bug Tracking Systems / Issue Trackers ... 87
2.35.1. Adding Issue Numbers to Log Messages ... 87
2.35.2. Getting Information from the Issue Tracker ... 89
2.36. TortoiseGit's Settings ... 90
2.36.1. General Settings ... 90
2.36.2. Icon Overlay Settings ... 104
2.36.3. Network Settings ... 108

2.36.4. External Program Settings ... 111
2.36.5. Saved Data Settings ... 117
2.36.6. Git ... 118
2.36.7. Client Side Hook Scripts ... 122
2.36.8. TortoiseGitBlame Settings ... 128
2.36.9. TortoiseGitUDiff Settings ... 130
2.36.10. Advanced Settings ... 130
2.36.11. Exporting TortoiseGit Settings ... 135
2.37. Working with worktrees ... 136
2.37.1. Creating a worktree ... 136
2.38. git svn dcommit ... 136
2.39. Git LFS Locking ... 136
2.39.1. Setting up the repository ... 136
2.39.2. Locking a file ... 137
2.39.3. Unlocking a file ... 137
2.39.4. Show Locks Dialog ... 137
2.40. Final Step ... 138
3. The GitWCRev Program ... 139
3.1. The GitWCRev Command Line ... 139
3.2. Keyword Substitution ... 140
3.3. Keyword Example ... 142
3.4. COM interface ... 143
A. Frequently Asked Questions (FAQ) ... 148
B. IBugTraqProvider interface ... 149
B.1. Naming conventions ... 149
B.2. The IBugTraqProvider interface ... 149
B.3. The IBugTraqProvider2 interface ... 150
C. Useful Tips For Administrators ... 154
C.1. Deploy TortoiseGit via group policies ... 154
C.2. Redirect the upgrade check ... 154
C.3. Disable context menu entries ... 156
D. Automating TortoiseGit ... 159
D.1. TortoiseGit Commands ... 159
D.2. TortoiseGitIDiff Commands ... 162
E. Implementation Details ... 164
E.1. Icon Overlays ... 164
F. Tips and tricks for SSH/PuTTY ... 166
F.1. Introduction ... 166
F.1.1. How to use sessions ... 166
F.2. FAQ and examples section ... 166
F.2.1. How to use a default key for all SSH connections ... 166
F.2.2. How to connect to a SSH server on a different port ... 166
F.2.3. How to use two different SSH keys for the same user on the same host ... 167
G. Git Official Documentation ... 168
G.1. Git User Manual ... 168
G.1.1. Git User Manual ... 168
G.2. Git Tutorial ... 233
G.2.1. gittutorial(7) ... 233
G.2.2. gittutorial-2(7) ... 241
G.2.3. gitcore-tutorial(7) ... 246
G.2.4. gitcvs-migration(7) ... 266
G.2.5. giteveryday(7) ... 268
G.3. Git Command Reference ... 274
G.3.1. git(1) ... 274
G.3.2. git-add(1) ... 301
G.3.3. git-am(1) ... 307
G.3.4. git-annotate(1) ... 311
G.3.5. git-apply(1) ... 314

G.3.6. git-archimport(1) ... 319
G.3.7. git-archive(1) ... 320
G.3.8. git-bisect(1) ... 324
G.3.9. git-blame(1) ... 330
G.3.10. git-branch(1) ... 337
G.3.11. git-bugreport(1) ... 344
G.3.12. git-bundle(1) ... 345
G.3.13. git-cat-file(1) ... 349
G.3.14. git-check-attr(1) ... 354
G.3.15. git-check-ignore(1) ... 356
G.3.16. git-check-mailmap(1) ... 358
G.3.17. git-check-ref-format(1) ... 358
G.3.18. git-checkout-index(1) ... 360
G.3.19. git-checkout(1) ... 363
G.3.20. git-cherry-pick(1) ... 371
G.3.21. git-cherry(1) ... 375
G.3.22. git-citool(1) ... 377
G.3.23. git-clean(1) ... 378
G.3.24. git-clone(1) ... 380
G.3.25. git-column(1) ... 387
G.3.26. git-commit-graph(1) ... 389
G.3.27. git-commit-tree(1) ... 391
G.3.28. git-commit(1) ... 394
G.3.29. git-config(1) ... 403
G.3.30. git-count-objects(1) ... 499
G.3.31. git-credential(1) ... 499
G.3.32. git-credential-cache--daemon(1) ... 502
G.3.33. git-credential-cache(1) ... 502
G.3.34. git-credential-store(1) ... 503
G.3.35. git-cvsexportcommit(1) ... 505
G.3.36. git-cvsimport(1) ... 507
G.3.37. git-cvsserver(1) ... 510
G.3.38. git-daemon(1) ... 516
G.3.39. git-describe(1) ... 520
G.3.40. git-diagnose(1) ... 523
G.3.41. git-diff-files(1) ... 524
G.3.42. git-diff-index(1) ... 541
G.3.43. git-diff-tree(1) ... 558
G.3.44. git-diff(1) ... 586
G.3.45. git-difftool(1) ... 610
G.3.46. git-fast-export(1) ... 612
G.3.47. git-fast-import(1) ... 616
G.3.48. git-fetch-pack(1) ... 636
G.3.49. git-fetch(1) ... 638
G.3.50. git-filter-branch(1) ... 652
G.3.51. git-fmt-merge-msg(1) ... 660
G.3.52. git-for-each-ref(1) ... 662
G.3.53. git-for-each-repo(1) ... 669
G.3.54. git-format-patch(1) ... 670
G.3.55. git-fsck-objects(1) ... 686
G.3.56. git-fsck(1) ... 686
G.3.57. git-fsmonitor--daemon(1) ... 693
G.3.58. git-gc(1) ... 695
G.3.59. git-get-tar-commit-id(1) ... 700
G.3.60. git-grep(1) ... 700
G.3.61. git-gui(1) ... 706
G.3.62. git-hash-object(1) ... 707
G.3.63. git-help(1) ... 708

G.3.64. git-hook(1) 711
G.3.65. git-http-backend(1) 712
G.3.66. git-http-fetch(1) 716
G.3.67. git-http-push(1) 717
G.3.68. git-imap-send(1) 719
G.3.69. git-index-pack(1) 722
G.3.70. git-init-db(1) 724
G.3.71. git-init(1) 724
G.3.72. git-instaweb(1) 727
G.3.73. git-interpret-trailers(1) 728
G.3.74. git-log(1) 735
G.3.75. git-ls-files(1) 778
G.3.76. git-ls-remote(1) 783
G.3.77. git-ls-tree(1) 785
G.3.78. git-mailinfo(1) 788
G.3.79. git-mailsplit(1) 790
G.3.80. git-maintenance(1) 791
G.3.81. git-merge-base(1) 797
G.3.82. git-merge-file(1) 800
G.3.83. git-merge-index(1) 802
G.3.84. git-merge-one-file(1) 803
G.3.85. git-merge-tree(1) 803
G.3.86. git-merge(1) 807
G.3.87. git-mergetool--lib(1) 821
G.3.88. git-mergetool(1) 822
G.3.89. git-mktag(1) 828
G.3.90. git-mktree(1) 829
G.3.91. git-mv(1) 829
G.3.92. git-multi-pack-index(1) 830
G.3.93. git-name-rev(1) 832
G.3.94. git-notes(1) 834
G.3.95. git-p4(1) 840
G.3.96. git-pack-objects(1) 851
G.3.97. git-pack-redundant(1) 857
G.3.98. git-pack-refs(1) 858
G.3.99. git-patch-id(1) 859
G.3.100. git-prune-packed(1) 860
G.3.101. git-prune(1) 861
G.3.102. git-pull(1) 862
G.3.103. git-push(1) 876
G.3.104. git-quiltimport(1) 890
G.3.105. git-range-diff(1) 891
G.3.106. git-read-tree(1) 895
G.3.107. git-rebase(1) 900
G.3.108. git-receive-pack(1) 921
G.3.109. git-reflog(1) 925
G.3.110. git-remote-ext(1) 926
G.3.111. git-remote-fd(1) 928
G.3.112. git-remote(1) 929
G.3.113. git-repack(1) 933
G.3.114. git-replace(1) 936
G.3.115. git-replay(1) 939
G.3.116. git-request-pull(1) 954
G.3.117. git-rerere(1) 955
G.3.118. git-reset(1) 957
G.3.119. git-restore(1) 964
G.3.120. git-rev-list(1) 967
G.3.121. git-rev-parse(1) 994

G.3.122. git-revert(1) 1007
G.3.123. git-rm(1) 1010
G.3.124. git-send-email(1) 1012
G.3.125. git-send-pack(1) 1021
G.3.126. git-sh-i18n--envsubst(1) 1024
G.3.127. git-sh-i18n(1) 1024
G.3.128. git-sh-setup(1) 1025
G.3.129. git-shell(1) 1026
G.3.130. git-shortlog(1) 1027
G.3.131. git-show-branch(1) 1039
G.3.132. git-show-index(1) 1042
G.3.133. git-show-ref(1) 1043
G.3.134. git-show(1) 1045
G.3.135. git-sparse-checkout(1) 1071
G.3.136. git-stage(1) 1077
G.3.137. git-stash(1) 1077
G.3.138. git-status(1) 1083
G.3.139. git-stripspace(1) 1090
G.3.140. git-switch(1) 1092
G.3.141. git-submodule(1) 1096
G.3.142. git-svn(1) 1102
G.3.143. git-symbolic-ref(1) 1118
G.3.144. git-tag(1) 1119
G.3.145. git-unpack-file(1) 1124
G.3.146. git-unpack-objects(1) 1125
G.3.147. git-update-index(1) 1126
G.3.148. git-update-ref(1) 1133
G.3.149. git-update-server-info(1) 1136
G.3.150. git-upload-archive(1) 1136
G.3.151. git-upload-pack(1) 1137
G.3.152. git-var(1) 1139
G.3.153. git-verify-commit(1) 1140
G.3.154. git-verify-pack(1) 1141
G.3.155. git-verify-tag(1) 1142
G.3.156. git-version(1) 1142
G.3.157. git-web--browse(1) 1143
G.3.158. git-whatchanged(1) 1145
G.3.159. git-worktree(1) 1145
G.3.160. git-write-tree(1) 1152
G.3.161. scalar(1) 1153
G.4. Misc 1155
G.4.1. gitcli(7) 1155
G.4.2. gitattributes(5) 1158
G.4.3. gitcredentials(7) 1174
G.4.4. gitdiffcore(7) 1178
G.4.5. gitignore(5) 1182
G.4.6. gitfaq(7) 1185
G.4.7. githooks(5) 1191
G.4.8. gitk(1) 1200
G.4.9. gitmailmap(5) 1203
G.4.10. gitmodules(5) 1205
G.4.11. gitnamespaces(7) 1206
G.4.12. gitremote-helpers(7) 1208
G.4.13. gitrepository-layout(5) 1215
G.4.14. gitrevisions(7) 1221
G.4.15. gitsubmodules(7) 1227
G.4.16. gitweb(1) 1231
G.4.17. gitweb.conf(5) 1241

G.4.18. gitworkflows(7) 1254
G.4.19. gitglossary(7) 1260
G.5. File formats, protocols and other developer interfaces 1270
G.5.1. gitformat-bundle(5) 1270
G.5.2. gitformat-chunk(5) 1272
G.5.3. gitformat-commit-graph(5) 1273
G.5.4. gitformat-index(5) 1276
G.5.5. gitformat-pack(5) 1283
G.5.6. gitformat-signature(5) 1293
G.5.7. gitprotocol-capabilities(5) 1296
G.5.8. gitprotocol-common(5) 1301
G.5.9. gitprotocol-http(5) 1303
G.5.10. gitprotocol-pack(5) 1310
G.5.11. gitprotocol-v2(5) 1320
Glossary 1332
Index 1335

List of Figures

2.1. Explorer showing icon overlays 5
2.2. Context menu for a directory under version control 6
2.3. Explorer file menu for a shortcut in a versioned folder 7
2.4. Right drag menu for a directory under version control 7
2.5. Create repository dialog 9
2.6. Successful repository creation message 9
2.7. Clone dialog 10
2.8. The Switch/Checkout dialog 11
2.9. The Commit dialog 13
2.10. The Commit Dialog Spellchecker 16
2.11. The Progress dialog showing a commit in progress 18
2.12. Explorer showing icon overlays 18
2.13. Explorer property page, Git tab 20
2.14. Check for Modifications 21
2.15. Pull dialog 23
2.16. Fetch dialog 24
2.17. Push dialog 25
2.18. Sync dialog 27
2.19. A running daemon dialog 28
2.20. Browse References Dialog dialog 29
2.21. Delete remote tags dialog 29
2.22. The add submodule dialog 30
2.23. Submodule context menu entries 30
2.24. The update submodule dialog 31
2.25. Button for updating submodules in progress dialog 31
2.26. Commit dialog with Changelists 33
2.27. The Revision Log Dialog 34
2.28. The Revision Log Dialog Top Pane with Context Menu 36
2.29. Collapsed revisions 37
2.30. The Search Log Messages Dialog 38
2.31. Top Pane Context Menu for 2 Selected Revisions 39
2.32. The Log Dialog Bottom Pane with Context Menu 40
2.33. Option Menu for the Log Filter 42
2.34. Commits-by-Author Histogram 44
2.35. Commits-by-Author Pie Chart 45
2.36. Commits-by-date Graph 46
2.37. A Revision Graph 47
2.38. RefLog Dialog 49
2.39. The Repository Browser 50
2.40. The Compare Revisions Dialog 53
2.41. The submodule difference dialog 54
2.42. The image difference viewer 56
2.43. Explorer context menu for unversioned files 58
2.44. Add finished 59
2.45. Explorer context menu for unversioned files 59
2.46. Ignore dialog 60
2.47. Explorer context menu for versioned files 61
2.48. Right drag menu for a directory under version control 63
2.49. Revert dialog 64
2.50. Clean dialog 66
2.51. The Reset dialog 67
2.52. The Abort Merge dialog 67
2.53. Stash changes dialog 68
2.54. (un)stash options 68
2.55. Bisect start 69

2.56. Bisect options 69
2.57. The Branch Dialog 70
2.58. The Tag Dialog 71
2.59. Merge dialog 73
2.60. Cherry Pick dialog 74
2.61. Rebase dialog 75
2.62. The resolve conflicts dialog 77
2.63. Resolve delete-modify conflict Dialog 78
2.64. Resolve submodule conflict Dialog 79
2.65. The Create Patch dialog 80
2.66. The Send Patches Dialog 81
2.67. The Choose Repository Dialog 82
2.68. The Apply Patch Dialog 83
2.69. The Request Pull Dialog 84
2.70. TortoiseGitBlame 85
2.71. The Export Dialog 86
2.72. Example issue tracker query dialog 90
2.73. The Settings Dialog, General Page 91
2.74. The Settings Dialog, Context Menu Page 93
2.75. The Settings Dialog, Context Menu 2 94
2.76. The Settings Dialog, Dialogs Page 95
2.77. Example of Symbolize ref names 96
2.78. The Settings Dialog, Dialogs Page 2 97
2.79. The Settings Dialog, Dialogs 3 Page 99
2.80. The Settings Dialog, colors Page 101
2.81. The Settings Dialog, colors Page 102
2.82. The Settings Dialog, colors Page 103
2.83. The Settings Dialog, Icon Overlays Page 104
2.84. The Settings Dialog, Icon Set Page 107
2.85. The Settings Dialog, Icon Handlers Page 108
2.86. The Settings Dialog, Network Page 109
2.87. The Settings Dialog, email settings 110
2.88. The Settings Dialog, Diff Viewer Page 111
2.89. The Settings Dialog, Merge Tool Page 113
2.90. The Settings Dialog, Diff/Merge Advanced Dialog 115
2.91. The Settings Dialog, Alternative editor Page 116
2.92. The Settings Dialog, Saved Data Page 117
2.93. The Settings Dialog, Git 119
2.94. The Settings Dialog, Git, Remote 120
2.95. The Settings Dialog, Git, Credential 121
2.96. The Settings Dialog, Hook Scripts Page 123
2.97. The Settings Dialog, Configure Hook Scripts 124
2.98. The Settings Dialog, Issue Tracker Integration Page 126
2.99. The Settings Dialog, Issue Tracker Config 127
2.100. The Settings Dialog, TortoiseGitBlame Page 128
2.101. The Settings Dialog, TortoiseGitUDiff Page 130
2.102. Taskbar with default grouping 132
2.103. Taskbar with repository grouping 132
2.104. Taskbar grouping with repository color overlays 133
2.105. Locks Dialog 137
C.1. The upgrade dialog 155

List of Tables

3.1. List of available command line switches 140
3.2. List of GitWCRev error codes 140
3.3. List of available keywords 140
3.4. COM/automation methods supported 143
C.1. Menu entries and their values 156
D.1. List of available commands and options 159
D.2. List of available options 162

List of Examples

G.1. Merge upwards 1255
G.2. Topic branches 1256
G.3. Merge to downstream only at well-defined points 1256
G.4. Throw-away integration branches 1256
G.5. Verify *master* is a superset of *maint* 1257
G.6. Release tagging 1257
G.7. Copy maint 1257
G.8. Update maint to new release 1257
G.9. Rewind and rebuild next 1258
G.10. Push/pull: Publishing branches/topics 1258
G.11. Push/pull: Staying up to date 1259
G.12. Push/pull: Merging remote topics 1259
G.13. format-patch/am: Publishing branches/topics 1259
G.14. format-patch/am: Keeping topics up to date 1259
G.15. format-patch/am: Importing patches 1259

Preface

- Do you work in a team?
- Has it ever happened that you were working on a file, and someone else was working on the same file at the same time? Did you lose your changes to that file because of that?
- Have you ever saved a file, and then wanted to revert the changes you made? Have you ever wished you could see what a file looked like some time ago?
- Have you ever found a bug in your project and wanted to know when that bug got into your files?

If you answered "yes" to one of these questions, then TortoiseGit is for you! Just read on to find out how TortoiseGit can help you in your work. It's not that difficult.

1. Audience

This book is written for computer literate folk who want to use Git to manage their data, but are uncomfortable using the command line client to do so. Since TortoiseGit is a windows shell extension it's assumed that the user is familiar with the windows explorer and knows how to use it.

2. Reading Guide

This Preface explains a little about the TortoiseGit project, the community of people who work on it, and the licensing conditions for using it and distributing it.

The Chapter 1, *Introduction* explains what TortoiseGit is, what it does, where it comes from and the basics for installing it on your PC.

If you need a general introduction to version control with *Git*, then we recommend two videos on YouTube: *Tech Talk: Linus Torvalds on git* [https://www.youtube.com/watch?v=4XpnKHJAok8] (about design and differences to other VCS) and *Tech Talk: Git* [https://www.youtube.com/watch?v=8dhZ9BXQgc4] (more technical). You can also read *Pro Git book (multiple translations as well as downloadable versions available)* [https://git-scm.com/book], Section G.1, "Git User Manual", or Section G.2.1, "gittutorial(7)" which are a short introductions to the *Git* revision control system, explain the different approaches to version control, and how Git works (with a bunch of examples).

The Chapter 2, *TortoiseGit Daily Use Guide* is the most important section as it explains all the main features of TortoiseGit and how to use them. It takes the form of a tutorial, starting with checking out a working tree, modifying it, committing your changes, etc. It then progresses to more advanced topics.

The section on Appendix D, *Automating TortoiseGit* shows how the TortoiseGit GUI dialogs can be called from the command line. This is useful for scripting where you still need user interaction.

The Section G.3.1, "git(1)" give git official document about command line client `git.exe`.

3. TortoiseGit is free!

TortoiseGit is free. You don't have to pay to use it, and you can use it any way you want. It is developed under the GNU General Public License (GPL).

TortoiseGit is an Open Source project. That means you have full read access to the source code of this program. Project Home is *https://tortoisegit.org/* [https://tortoisegit.org/]

4. Community

Both TortoiseGit and Git are developed by a community of people who are working on those projects. They come from different countries all over the world and joined together to create wonderful programs.

5. Acknowledgments

Frank Li "lznuaa@gmail.com"

for founding the TortoiseGit project

Sven Strickroth "email@cs-ware.de"

for the hard work to get TortoiseGit to what it is now, and his leadership of the project

Sup Yut Sum "ch3cooli@gmail.com"

for bug reports and lots of improvements (code and translations)

Yue Lin Ho "b8732003@student.nsysu.edu.tw"

for bug reports, work on the mailing list and lots of improvements (code and translations)

myagi (Georg Fischer) "snowcoder@gmail.com"

For hard work to get TortoiseGit Overlay work.

Colin Law

Johan't Hart

Laszlo Papp "djszapi@archlinux"

Tim Kemp

for founding the TortoiseSVN project (TortoiseGit is based on this project)

Stefan Küng

for the hard work on TortoiseSVN

Lübbe Onken

for the beautiful icons, logo, bug hunting and translating

Simon Large

for helping with the documentation and bug hunting on TortoiseSVN

The Tigris Style project

for some of the styles which are reused in this documentation

Our Contributors

for the patches, bug reports and new ideas, and for helping others by answering questions on our mailing list.

Our Donators

6. Terminology used in this document

To make reading the docs easier, the names of all the screens and Menus from TortoiseGit are marked up in a different font. The Log Dialog for instance.

A menu choice is indicated with an arrow. TortoiseGit → Show Log means: select *Show Log* from the *TortoiseGit* context menu.

Where a local context menu appears within one of the TortoiseGit dialogs, it is shown like this: Context Menu → Save As ...

User Interface Buttons are indicated like this: Press OK to continue.

User Actions are indicated using a bold font. **Alt+A**: press the **Alt**-Key on your keyboard and while holding it down press the **A**-Key as well. Right-drag: press the right mouse button and while holding it down *drag* the items to the new location.

System output and keyboard input is indicated with a `different` font as well.

Important

Important notes are marked with an icon.

Tip

Tips that make your life easier.

Caution

Places where you have to be careful what you are doing.

Warning

Where extreme care has to be taken, data corruption or other nasty things may occur if these warnings are ignored.

Chapter 1. Introduction

Version control is the art of managing changes to information. It has long been a critical tool for programmers, who typically spend their time making small changes to software and then undoing or checking some of those changes the next day. Imagine a team of such developers working concurrently - and perhaps even simultaneously on the very same files! - and you can see why a good system is needed to *manage the potential chaos*.

1.1. What is TortoiseGit?

TortoiseGit is a free open-source client for the *Git* version control system. That is, TortoiseGit manages files over time. Files are stored in a local *repository*. The repository is much like an ordinary file server, except that it remembers every change ever made to your files and directories. This allows you to recover older versions of your files and examine the history of how and when your data changed, and who changed it. This is why many people think of Git and version control systems in general as a sort of “time machine”.

Some version control systems are also software configuration management (SCM) systems. These systems are specifically tailored to manage trees of source code, and have many features that are specific to software development - such as natively understanding programming languages, or supplying tools for building software. Git, however, is not one of these systems; it is a general system that can be used to manage *any* collection of files, including source code.

Git is an *open source, distributed version control system* designed to handle everything from small to very large projects with speed and efficiency. Every Git clone is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server. Branching and merging are fast and easy to do.

1.2. TortoiseGit's History

In 2008, Frank Li found that Git was a very good version control system, but it lacked a good GUI client. The idea for a Git client as a Windows shell integration was inspired by the similar client for SVN named TortoiseSVN.

Frank studied the source code of TortoiseSVN and used it as a base for TortoiseGit. He then started the project, registered the project at code.google.com and put the source code online.

At the end of 2010 Sven Strickroth joined the TortoiseGit project. Then, he became the current maintainer few years later.

From August 2015, GoogleCode was shut down and the TortoiseGit project established their website TortoiseGit.org and migrated the main repository and issue tracker to GitLab.

As Git became more stable it attracted more and more users who also started using TortoiseGit as their Git client.

For more information what changed over the releases check out the latest release notes [https://tortoisegit.org/releasenotes] or inspect our git commit history [https://tortoisegit.org/sourcecode].

1.3. TortoiseGit's Features

What makes TortoiseGit such a good Git client? Here's a short list of features.

Shell integration

TortoiseGit integrates seamlessly into the Windows shell (i.e. the explorer). This means you can keep working with the tools you're already familiar with. And you do not have to change into a different application each time you need functions of the version control!

And you are not even forced to use the Windows Explorer. TortoiseGit's context menus work in many other file managers, and in the File/Open dialog which is common to most standard Windows applications. You should, however, bear in mind that TortoiseGit is intentionally developed as extension for the Windows Explorer. Thus it is possible that in other applications the integration is not as complete and e.g. the icon overlays may not be shown.

Icon overlays

The status of every versioned file and folder is indicated by small overlay icons. That way you can see right away what the status of your working tree is.

The icon overlays are based on TortoiseOverlays (https://tortoisesvn.net [https://tortoisesvn.net/])

Easy access to Git commands

All Git commands are available from the explorer context menu. TortoiseGit adds its own submenu there.

Since TortoiseGit is a Git client, we would also like to show you some of the features of Git itself:

Distributed version control

Like most other modern version control systems, Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch. Repositories can be easily accessed via the efficient Git protocol (optionally wrapped in SSH for authentication and security) or simply using HTTP - you can publish your repository anywhere without any special web server configuration required.

Atomic commits

A commit either goes into the repository completely, or not at all.

Strong support for non-linear development

Git supports rapid and convenient branching and merging, and includes powerful tools for visualizing and navigating a non-linear development history.

Efficient handling of large projects

Git is very fast and scales well even when working with large projects and long histories. It is commonly an order of magnitude faster than most other version control systems, and several orders of magnitude faster on some operations. It also uses an extremely efficient packed format for long-term revision storage that currently tops any other open source version control system.

Cryptographic authentication of history

The Git history is stored in such a way that the name of a particular revision (a "commit" in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed. Also, tags can be cryptographically signed.

Efficient branching and tagging

The cost of branching and tagging need not be proportional to the project size. Branch is just head of commits. Tag is friend name of commit hash.

Toolkit design

Following the Unix tradition, Git is a collection of many small tools written in C, and a number of scripts that provide convenient wrappers. Git provides tools for both easy human usage and easy scripting to perform new clever operations.

1.4. Installing TortoiseGit

1.4.1. System requirements

TortoiseGit runs on Windows 7 or higher. Windows 98, Windows ME, Windows NT4, Windows 2000, Windows XP and Windows Vista are no longer supported. If you are running such an old system, you can still use older, however unsupported, releases of TortoiseGit. Those can be found on the download server [https://download.tor-

toisegit.org/] (TortoiseGit 1.7 dropped support for Windows 2000; TortoiseGit 1.9 dropped support for Windows XP, TortoiseGit 2.5 dropped support for Windows Vista).

If you encounter any problems during or after installing TortoiseGit please refer to Appendix A, *Frequently Asked Questions (FAQ)* first.

1.4.2. Installation

TortoiseGit comes with an easy to use installer. Double click on the installer file and follow the instructions. The installer will take care of the rest.

One prerequisite of TortoiseGit is that it requires an already installed (command line) Git client which provides a `git.exe`. Git for Windows [https://gitforwindows.org/] is recommended (Cygwin and MSYS2 Git also work, see Section 2.36.1, "General Settings" for configuration. Please note that Cygwin and MSYS2 Git are not officially supported by TortoiseGit as the developers only use Git for Windows. Bug reports, however, are welcome). Installation of Git for Windows can be done with preselected options, however, no need to install the "Windows Explorer integration". If you know about CRLF and LF line endings and you have editors coping with that, you should select "Checkout as-is, commit as-is" in order to prevent automatic translations.

Important

You need Administrator privileges to install TortoiseGit.

1.4.3. Language Packs

The TortoiseGit user interface has been translated into many different languages, so you may be able to download a language pack to suit your needs. You can find the language packs on our *translation status page* [https://tortoisegit.org/download]. And if there is no language pack available yet, why not join the team and submit your own translation [https://tortoisegit.org/translate] ;-)

Each language pack is packaged as a `.msi` installer. Just run the install program after the installation of the main TortoiseGit package and follow the instructions. After the installation finishes, the translation will be available and can be selected in settings dialog (cf. Section 2.36.1, "General Settings").

1.4.4. Spell checker

TortoiseGit includes a spell checker which allows you to check your commit log messages. This is especially useful if the project language is not your native language. By default the spell checker uses the same dictionary files as *LibreOffice* [https://www.libreoffice.org/], *OpenOffice.org* [https://www.openoffice.org/] and *Mozilla* [https://mozilla.org]. TortoiseGit can also use the spell checker shipped with Windows 8+. However, this needs to be enabled manually in advanced settings (key `Win8SpellChecker`) at the moment.

The installer by default automatically adds the US English dictionary. If you want other languages, the easiest option is simply to install one of TortoiseGit's language packs (see Section 1.4.3, "Language Packs"). This will install the appropriate dictionary files as well as the TortoiseGit local user interface. After the installation finishes, the translation will be available. When using the Windows spell checker, you need to install the dictionary in Windows first.

Or you can install the dictionaries yourself. If you have OpenOffice.org, LibreOffice or Mozilla installed, you can copy those dictionaries, which are located in the installation folders for those applications. Otherwise, you need to download the required dictionary files from *https://cgit.freedesktop.org/libreoffice/dictionaries/* [https://cgit.freedesktop.org/libreoffice/dictionaries/].

Once you have got the dictionary files, you probably need to rename them so that the filenames only have the locale chars in it. Example:

- `en_US.aff`
- `en_US.dic`

Then just copy them into the `%APPDATA%\TortoiseGit\dic` folder. If that folder isn't there, you have to create it first. TortoiseGit will also search the `Languages` sub-folder of the TortoiseGit installation folder (normally this will be `C:\Program Files\TortoiseGit\Languages`); this is the place where the language packs put their files. However, the `%APPDATA%`-folder doesn't require administrator privileges and, thus, has higher priority. The next time you start TortoiseGit, the spell checker will be available.

If you install multiple dictionaries, TortoiseGit uses these rules to select which one to use.

1. Check the `tgit.projectlanguage` setting. This setting can be set using TortoiseGit Settings Dialogs 3 page (Section 2.36.1.5, “TortoiseGit Dialog Settings 3”). Refer to Section G.3.29, “git-config(1)” for information about setting properties (use the `LCID Dec` value as assigned by Microsoft [https://docs.microsoft.com/openspecs/windows_protocols/ms-lcid/a9eac961-e77d-41a6-90a5-ce1a8b0cdb9c]).

2. If no project language is set, or that language is not installed, try the language corresponding to the Windows locale.

3. If the exact Windows locale doesn't work, try the “Base” language, e.g. `de_CH` (Swiss-German) falls back to `de_DE` (German).

4. If none of the above works, then the default language is English, which is included by default with the standard installation.

When using the Windows spell checker, this is tried first. If no Windows dictionary, based on these fallback-rules, could be found then the "old" spell checker is used as a fallback.

Chapter 2. TortoiseGit Daily Use Guide

This document describes day to day usage of the TortoiseGit client. It is *not* an introduction to version control systems, and *not* an introduction to Git. It is more like a place you may turn to when you know approximately what you want to do, but don't quite remember how to do it.

For hints where to find more information about doing version control with Git see Section 2, “Reading Guide”.

This document is also a work in progress, just as TortoiseGit and Git are. If you find any mistakes, please report them to the mailing list so we can update the documentation. Some of the screenshots in the Daily Use Guide (DUG) might not reflect the current state of the software. Please forgive us. We're working on TortoiseGit in our free time.

In order to get the most out of the Daily Use Guide:

- You should have installed TortoiseGit already.
- You should be familiar with version control systems.
- You should know the basics of Git.
- You should have set up a server and/or have access to a Git repository.

2.1. Getting Started

2.1.1. Icon Overlays

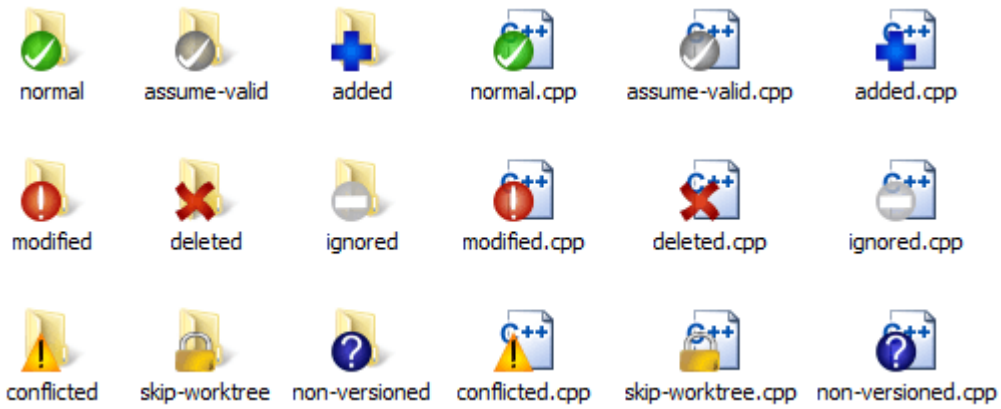


Figure 2.1. Explorer showing icon overlays

One of the most visible features of TortoiseGit is the icon overlays which appear on files in your working tree. These show you at a glance which of your files have been modified. Refer to Section 2.6.1, “Icon Overlays” to find out what the different overlays represent.

2.1.2. Context Menus

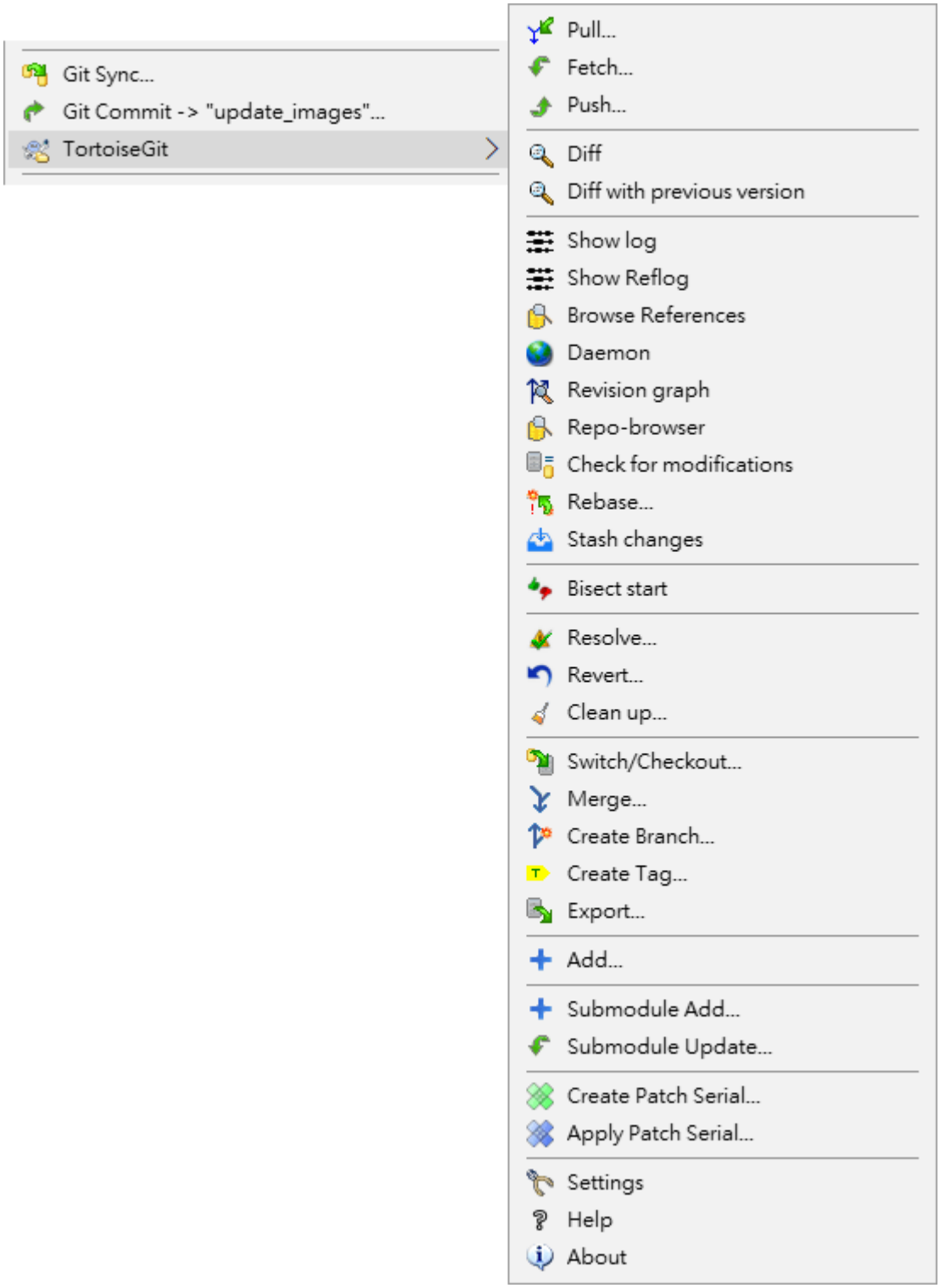


Figure 2.2. Context menu for a directory under version control

All TortoiseGit commands are invoked from the context menu of the windows explorer. Most are directly visible, when you right click on a file or folder. The commands that are available depend on whether the file or folder or its parent folder is under version control or not. You can also see the TortoiseGit menu as part of the Explorer file menu.

Tip

Some commands which are very rarely used are only available in the extended context menu. To bring up the extended context menu, hold down the **Shift** key when you right-click.

In some cases you may see several TortoiseGit entries. This is not a bug!

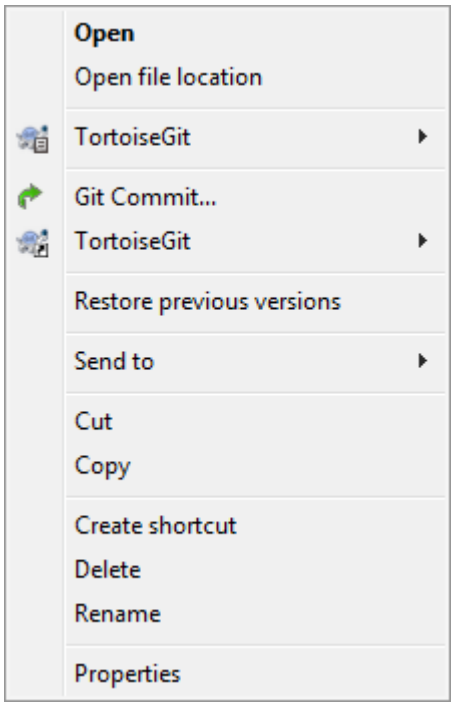


Figure 2.3. Explorer file menu for a shortcut in a versioned folder

This example is for an unversioned shortcut within a versioned folder, and in the Explorer file menu there are *two* entries for TortoiseGit. One for the shortcut itself and the second for the object the shortcut is pointing to. To help you distinguish between them, the icons have an indicator in the lower right corner to show whether the menu entry is for a file, a folder, a shortcut or for multiple selected items.

2.1.3. Drag and Drop

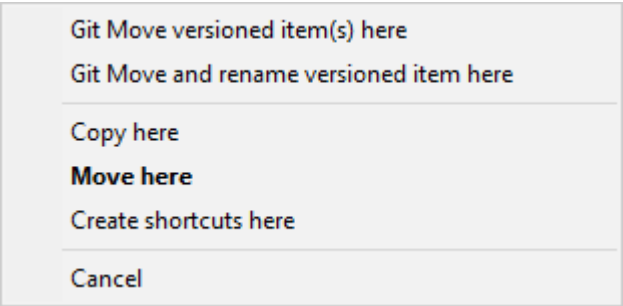


Figure 2.4. Right drag menu for a directory under version control

Other commands are available as drag handlers, when you right drag files or folders to a new location inside working trees or when you right drag a non-versioned file or folder into a directory which is under version control.

2.1.4. Common Shortcuts

Some common operations have well-known Windows shortcuts, but do not appear on buttons or in menus. If you can't work out how to do something obvious, like refreshing a view, check here.

F1

Help, of course.

F5

Refresh the current view. This is perhaps the single most useful one-key command. For example ... In Explorer this will refresh the icon overlays on your working tree. In the commit dialog it will re-scan the working tree to see what may need to be committed. In the Revision Log dialog it will contact the repository again to check for more recent changes.

Ctrl-A

Select all. This can be used if you get an error message and want to copy and paste into an email. Use Ctrl-A to select the error message and then ...

Ctrl-C

... Copy the selected text.

Ctrl-F

Search

2.1.5. Authentication

2.1.5.1. SSH (URLs look like git@example.com)

TortoiseGitPlink is recommended as SSH client because it better integrates with Windows. By default TortoiseGitPlink does not store passwords, you can use the PuTTY authentication agent for caching the password (done automatically if a PuTTY key is configured for a remote). For advanced tips & tricks see Appendix F, *Tips and tricks for SSH/PuTTY*. Note, however, that TortoiseGitPlink does not respect `~/.ssh/config` which is OpenSSH specific (see PuTTY tips & tricks or configure OpenSSH as SSH client, see next paragraph). If you also want to use TortoiseGitPlink on Git Bash, create an environment variable called `GIT_SSH` with the path to the PuTTY plink.exe or preferably to TortoiseGitPlink.exe. This can be done by re-executing the Git for Windows installer (there you can choose which SSH client to use), on the command line by executing `set GIT_SSH=PATH_TO_PLINK.EXE"` (`C:\Program Files\TortoiseGit\bin\TortoiseGitPlink.exe` on default installations) or configure the environment variables *permanently* [https://www.computerhope.com/issues/ch000549.htm].

It is also possible to use OpenSSH (shipped with Git for Windows, Cygwin, and MSYS2). Just open TortoiseGit settings and open the Network page and enter `ssh.exe` as SSH client, see Section 2.36.3, "Network Settings" and *this answer on StackOverflow* [https://stackoverflow.com/a/32115724/3906760]. When OpenSSH is used, you can also make use of `~/.ssh/config` (cf. *this answer on StackOverflow* [https://stackoverflow.com/q/30320458/3906760]).

2.1.5.2. HTTP/HTTPS (URLs start with https:// or http://)

By default Git does not save/cache credentials. However, you can configure a *credential helper* [https://stackoverflow.com/q/14000173/3906760] (recommended, also see Section G.4.3, "gitcredentials(7)") or manually use *`%HOME%/_netrc`* [https://stackoverflow.com/revisions/6031266/6].

If you have set up a credential store and you want to clear some stored credentials see *this answer on StackOverflow* [https://stackoverflow.com/a/31782500/3906760].

2.1.6. Maximizing Windows

Many of TortoiseGit's dialogs have a lot of information to display, but it is often useful to maximize only the height, or only the width, rather than maximizing to fill the screen. As a convenience, there are shortcuts for this on the Maximize button. Use the middle mouse button to maximize vertically, and right mouse to maximize horizontally.

2.2. Create Repository

This section talks about how to create a git repository. Creating an empty git repository is very simple. At an empty directory, just use the explorer context menu and select Git Create Repository here .

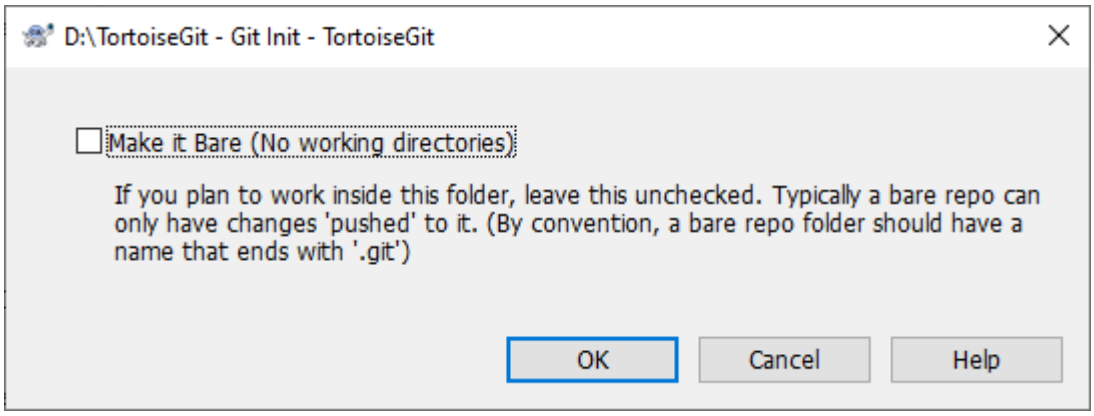


Figure 2.5. Create repository dialog

You can choose here between a bare and normal git repository. A normal repository has a working tree attached to which files can be checkout out and committed whereas a bare repository only can be pushed to and pulled from. After a (non bare) repository is created a message box will be shown:

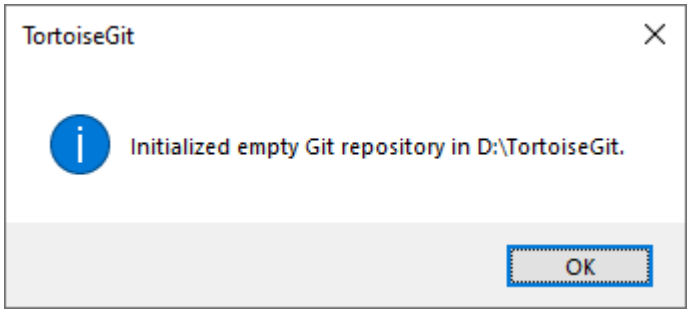


Figure 2.6. Successful repository creation message

You can find more information at Section G.3.71, “git-init(1)”.

2.3. Clone Repository

This section talks about how to clone a git repository from an existing repository. This operation is used to get a full copy of a remote repository. Cloning a git repository is very simple. At an empty directory, just use the explorer context menu and select Git Clone....

The Clone Dialog will show.

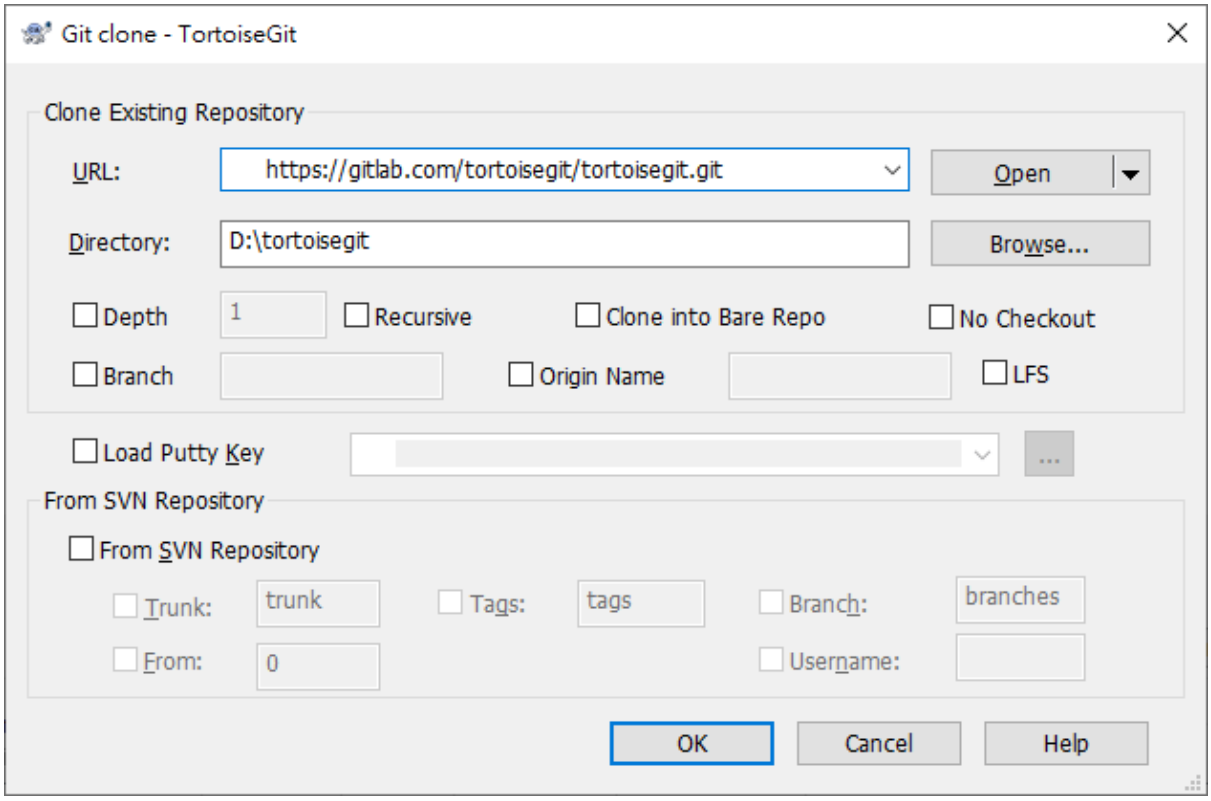


Figure 2.7. Clone dialog

URL: Input repository URL address, which you will clone *from*. You can click Browse to browse directory.

Directory: Input your local directory, which you will clone *to*. You can click Browse to browse directory.

Depth: Create a shallow clone with a limited history cut at the specified number of commits.

Recursive: Submodules are initialized and cloned using their default settings.

Clone into Bare Repo: Clone into a bare Git repository without a working tree.

No Checkout: No checkout of HEAD is performed after cloning is completed. This will result in an empty working tree.

Branch: Instead of pointing the newly created HEAD to the cloned repository's HEAD, point to the specified branch instead.

Origin Name: Instead of using the remote name "origin" (default) to keep track of the upstream repository, use the specified name.

LFS: Use Git LFS (Large File Storage) - this option is only available if Git LFS is installed.

If you check the Load Putty Key checkbox, clone will auto load putty key file with Pageant. You can click ... to browse for a putty key file.

Clone will checkout current HEAD to work space automatically.

Git clone supports HTTP, Git and SSH protocol. Section 2.36.3, "Network Settings" shows how to choose SSH client. OpenSSH, Plink or TortoiseGitPlink.

You can find more information at Section G.3.24, "git-clone(1)"

2.4. Checking Out A Working Tree (Switch to commit)

The Switch/Checkout dialog can be used to checkout a specific version to the working tree (i.e., all files are updated to match their state of the selected version). Normally, a specific version will be represented by a (local) branch which is set as the current branch (cf. Section 2.27, “Branching/Tagging” and Section 2, “Repositories and Branches”).

Select a git repository directory in windows explorer Right click to pop up the context menu and select the command TortoiseGit → Switch/Checkout..., which brings up the following dialog box:

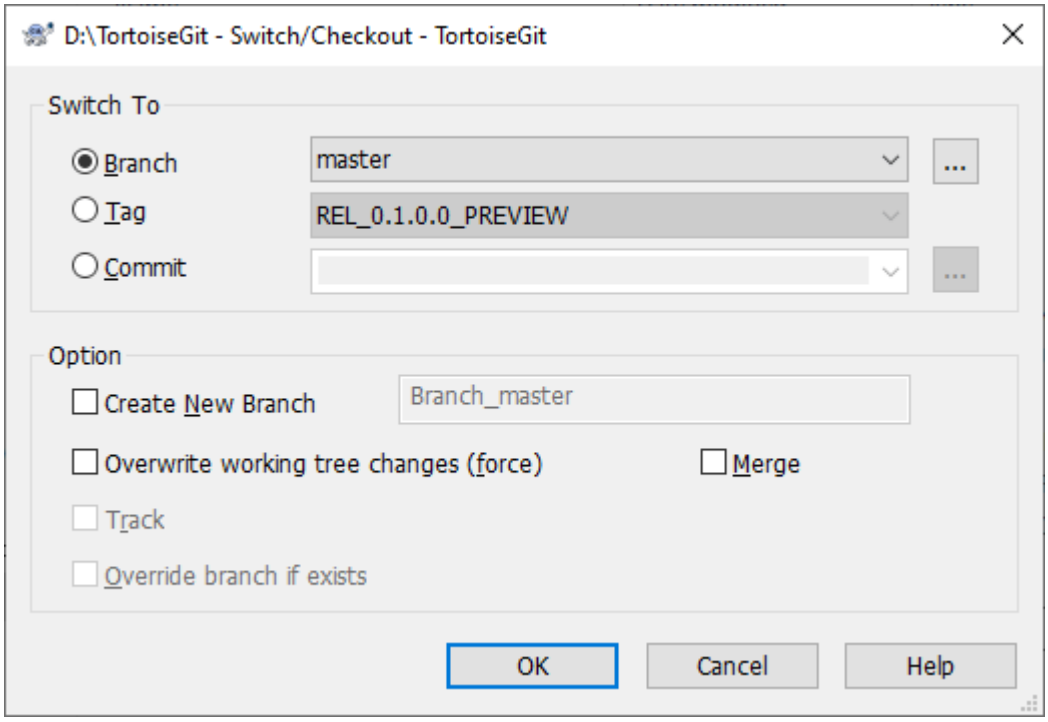


Figure 2.8. The Switch/Checkout dialog

If you enter a branch name at Create New Branch, a new branch will be created. Also, the new branch will be set as the current branch (HEAD).

You can click on the ... to browse the references/branches/log to choose a branch to checkout.

Check Overwrite working tree changes (force) will overwrite uncommitted changes in the working tree with the selected version.

When you selected a remote branch, you can check Track in order to track the remote branch. When you open the push, pull or sync dialog, the remote branch will be pre-selected automatically.

You can find more information at Section G.3.19, “git-checkout(1)”

Important

If you checkout/switch to a Tag or Commit, you should create a new branch. Otherwise you will work at "no branch" (detached HEAD state; i.e., there is no current branch, cf. the section called “DETACHED HEAD”). This can be easily fixed by creating a branch at this version and switching to it.

Exporting

Sometimes you may want to create a local copy without any of those `.git` directories, e.g. to create a zipped tarball of your source. Read Section 2.34, “Exporting a Git Working Tree” to find out how to do that.

2.5. Committing Your Changes To The Repository

Storing the changes you made to your working tree is known as *committing* the changes. you can use TortoiseGit → Check for Modifications first, to see which files have changed locally.

2.5.1. The Commit Dialog

If there are no conflicts, you are ready to commit your changes. Select any file and/or folders you want to commit, then TortoiseGit → Commit....

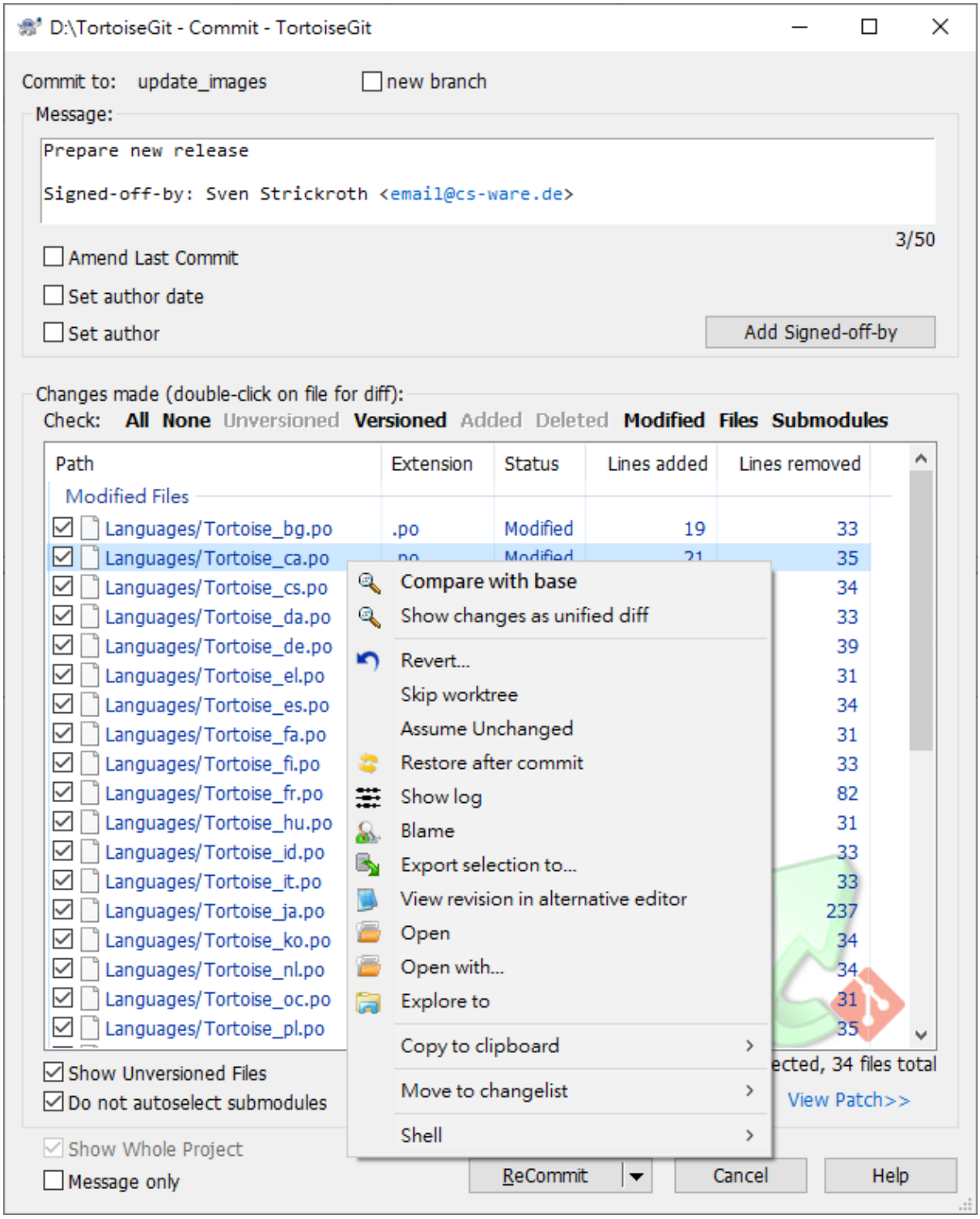


Figure 2.9. The Commit dialog

The commit dialog will show you every changed file, including added, deleted and unversioned files. If you don't want a changed file to be committed, just uncheck that file. If you want to include an unversioned file, just check that file to add it to the commit.

Default commit dialog just list select paths and their child directory files. If you want to list all files of project, you can just click Whole Project.

Many unversioned files in the commit dialog

If you think that the commit dialog shows you too many unversioned (e.g. compiler generated or editor backup) files, there are several ways to handle this. You can:

- add the file to the `.gitignore` list using TortoiseGit → Add to ignore list

Read Section 2.20, "Ignoring Files And Directories" for more information.

Double clicking on any modified file in the commit dialog will launch the external diff tool to show your changes. The context menu will give you more options, as shown in the screenshot. You can also drag files from here into another application such as a text editor or an IDE.

You can select or deselect items by clicking on the checkbox to the left of the item.

The columns displayed in the bottom pane are customizable. If you right click on any column header you will see a context menu allowing you to select which columns are displayed. You can also change column width by using the drag handle which appears when you move the mouse over a column boundary. These customizations are preserved, so you will see the same headings next time.
The color code of the different items is described in Section 2.6.3, "Status".

Please note that the commit dialog always shows staged files - even if it was started on a different folder (this is by design in order to not forget to commit those, e.g. in case of a merge). Also, in the lower left there is the checkbox Show whole project. Use this to override the file/folder filter and show all changed files for the whole repository. This setting is remembered for a repository - even if you started the commit dialog on a single file.

Drag and Drop

You can drag files into the commit dialog from elsewhere, as long as the working tree is the very same. For example, you may have a huge working tree with several explorer windows open to look at distant folders of the hierarchy. If you want to avoid committing from the top level folder (with a lengthy folder crawl to check for changes) you can open the commit dialog for one folder and drag in items from the other windows to include within the same atomic commit.

You can drag unversioned files which reside within a working tree into the commit dialog, and they will be Git added automatically.

Commits are just local

Please note, that all commits are just local and only affect your local working tree. In order to share them with others you need to push them to a remote repository. See Section 2.8, "Push" and Section 2.9, "Sync" for more information.

2.5.2. Change Lists

The commit dialog supports changelist feature to help with grouping related files together. Find out about this feature in Section 2.13, "Change Lists".

2.5.3. Excluding Items from the Commit List

Sometimes you have versioned files that change frequently but that you really don't want to commit. Sometimes this indicates a flaw in your build process - why are those files versioned? should you be using template files? But occasionally it is inevitable. A classic reason is that your IDE changes a timestamp in the project file every time you build. The project file has to be versioned as it includes all the build settings, but it doesn't need to be committed just because the timestamp changed.

To help out in awkward cases like this, there is a Git flag for files called `skip-worktree` - then files are treated as unmodified and Git also refuses to merge those on merge/pull (cf. the section called "SKIP-WORKTREE BIT"). As another way to tackle cases like this, we have reserved a changelist called `ignore-on-commit`. Any file added to this changelist will automatically be unchecked in the commit dialog. You can still commit changes, but you have to select it manually in the commit dialog.

2.5.4. Commit only parts of files

Sometimes you want to only commit parts of the changes you made to a file. Such a situation usually happens when you're working on something but then an urgent fix needs to be committed, and that fix happens to be in the same file you're working on.

right click on the file and use Context Menu → Restore after commit. This will create a copy of the file as it is. Then you can edit the file, e.g. in TortoiseGitMerge and undo all the changes you don't want to commit. After saving those changes you can commit the file.

Using TortoiseGitMerge

If you use TortoiseGitMerge to edit the file, you can either edit the changes as you're used to, or mark all the changes that you want to include. right click on a modified block and use Context Menu → Mark this block to include that change. Finally right click and use Context Menu → Use left file except marked blocks which will invert your changes (unmarked blocks) that you don't want to them to appear in current commit.

After the commit is done, the copy of the file is restored automatically, and you have the file with all your modifications that were not committed back.

2.5.5. Commit Log Messages

Be sure to enter a log message which describes the changes you are committing. This will help you to see what happened and when, as you browse through the project log messages at a later date. The message can be as long or as brief as you like; many projects have guidelines for what should be included, the language to use, and sometimes even a strict format.

You can apply simple formatting to your log messages using a convention similar to that used within emails. To apply styling to `text`, use `*text*` for bold, `_text_` for underlining, and `^text^` for italics.

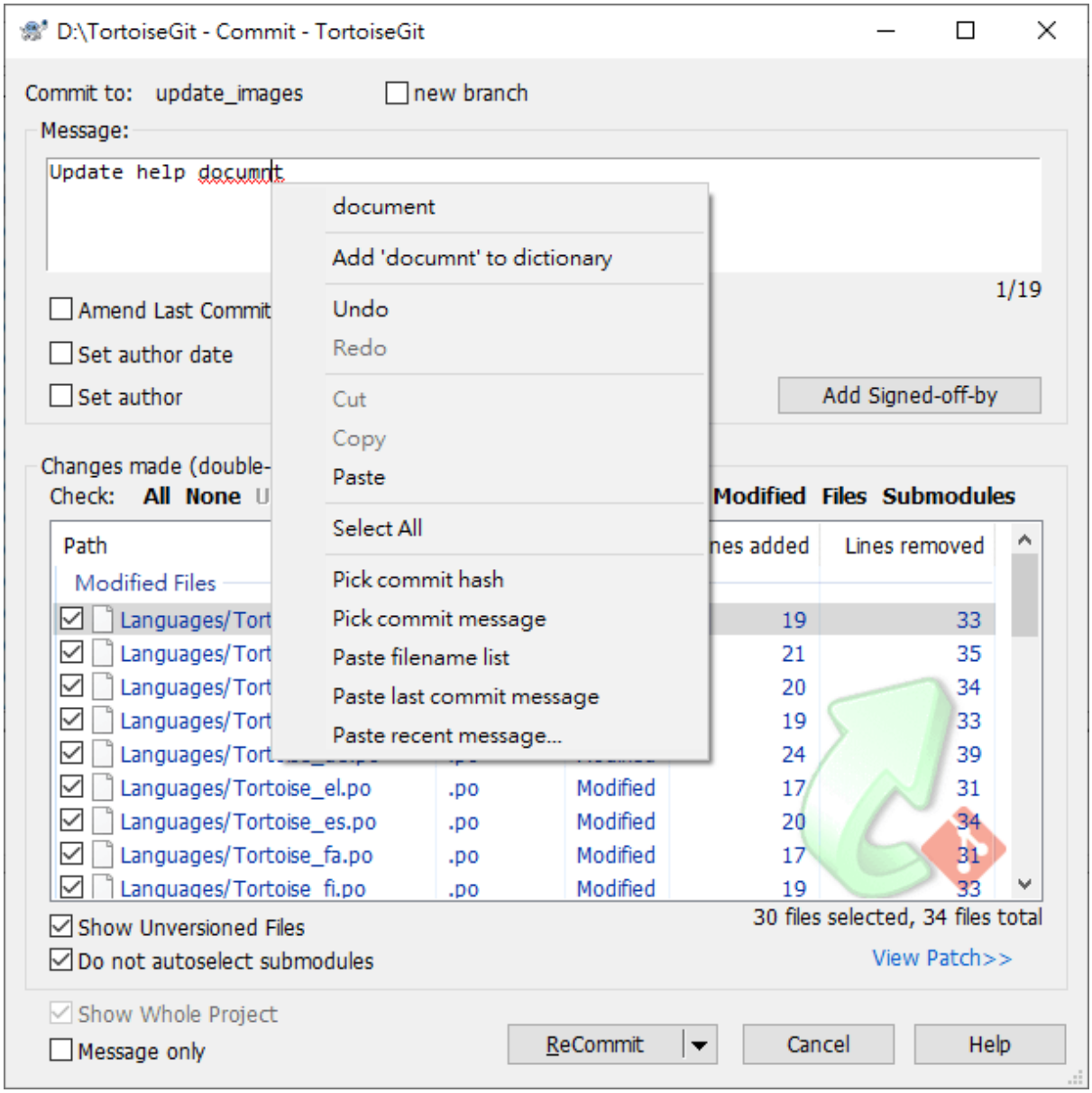


Figure 2.10. The Commit Dialog Spellchecker

TortoiseGit includes a spellchecker to help you get your log messages right (cf. Section 1.4.4, "Spell checker"). This will highlight any misspelled words. Use the context menu to access the suggested corrections. Of course, it doesn't know *every* technical term that you do, so correctly spelt words will sometimes show up as errors. But don't worry. You can just add them to your personal dictionary using the context menu.

The log message window also includes a filename and function auto-completion facility. This uses regular expressions to extract class and function names from the (text) files you are committing, as well as the filenames themselves. If a word you are typing matches anything in the list (after you have typed at least 3 characters, or pressed **Ctrl+Space**), a drop-down appears allowing you to select the full name. The regular expressions supplied with TortoiseGit are held in the TortoiseGit installation `bin` folder. You can also define your own regexes and store them in `%APPDATA%\TortoiseGit\autolist.txt`. Of course your private autolist will not be overwritten when you update your installation of TortoiseGit. If you are unfamiliar with regular expressions, take a look at the introduction at *https://en.wikipedia.org/wiki/Regular_expression* [https://en.wikipedia.org/wiki/Regular_expression], and the online documentation and tutorial at *https://www.regular-expressions.info/* [https://www.regular-expressions.info/].

Getting the regex just right can be tricky, so to help you sort out a suitable expression there is a test dialog which allows you to enter an expression and then type in filenames to test it against. Start it from the command prompt using the command `TortoiseGitProc.exe /command:autotexttest`.

You can re-use previously entered log messages. Just use the command Context Menu → Paste Recent messages to view a list of the last few messages you entered for this working tree. The number of stored messages can be customized in the TortoiseGit settings dialog.

The log message window also includes a commit message snippet facility. These snippets are shown in the autocomplete drop-down once you type a snippet shortcut, and selecting the snippet in the autocomplete drop-down then inserts the full text of the snippet. The snippets supplied with TortoiseGit are held in the TortoiseGit installation `bin` folder. You can also define your own snippets and store them in `%APPDATA%\TortoiseGit\snippet.txt`. # is the comment character. Use escape sequences `\t`, `\r`, `\n`, and `\\`.

You can add your name and email address to the end of the log message by clicking Add Signed-off-by.

You can clear all stored commit messages from the Saved data page of TortoiseGit's settings, or you can clear individual messages from within the Recent messages dialog using the **Delete** key.

If you want to include the checked paths in your log message, you can use the command Context Menu → Paste filename list in the edit control.

Another way to insert the paths into the log message is to simply drag the files from the file list onto the edit control.

Using keyboard

You can access the OK button from keyboard by pressing **Ctrl+return**.

Integration with Bug Tracking Tools

If you have activated the bug tracking system, you can set one or more Issues in the Bug-ID / Issue-No: text box. Multiple issues should be comma separated. Alternatively, if you are using regex-based bug tracking support, just add your issue references as part of the log message. Learn more in Section 2.35, “Integration with Bug Tracking Systems / Issue Trackers”.

Adjust the size of message text box

Move your mouse to the gap between "Message" group box and "Changes made" group box, then drag the separator.

Commit to a new branch

If you want to commit to a fresh branch (based on the current branch), you can check the new branch checkbox and enter a branch name in the displayed textbox.

Commit multiple times in a row and directly pushing changes

The main button Commit has a drop-down menu. There are the options ReCommit and Commit & push. The option ReCommit commits your changes and leaves the Commit dialog open, so that you can continue committing. The last option Commit & push will commit your changes and immediately push your changes. If no remote tracking branch is configured for the current active branch, the push dialog (cf. Section 2.8, “Push”) is opened.

2.5.6. Commit Progress

After pressing Commit, a dialog appears displaying the progress of the commit.

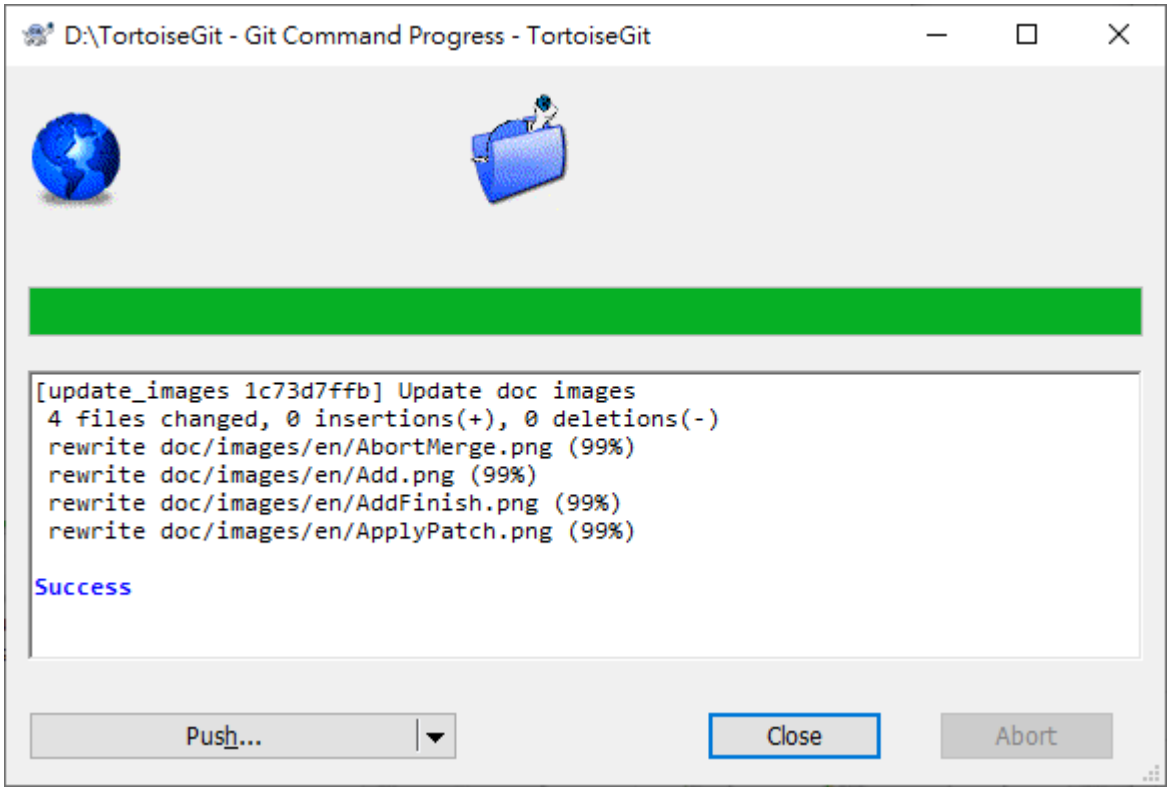


Figure 2.11. The Progress dialog showing a commit in progress

In the lower left, there is a menu button which provides shortcuts to further steps, such as ReCommit (resets the commit dialog and allows you to continue committing) or Push in order to push your commit to a remote repository.

You can find more information at Section G.3.28, “git-commit(1)”.

2.6. Getting Status Information

While you are working on your working tree you often need to know which files you have changed/added/removed or renamed, or even which files got changed and committed by others.

2.6.1. Icon Overlays

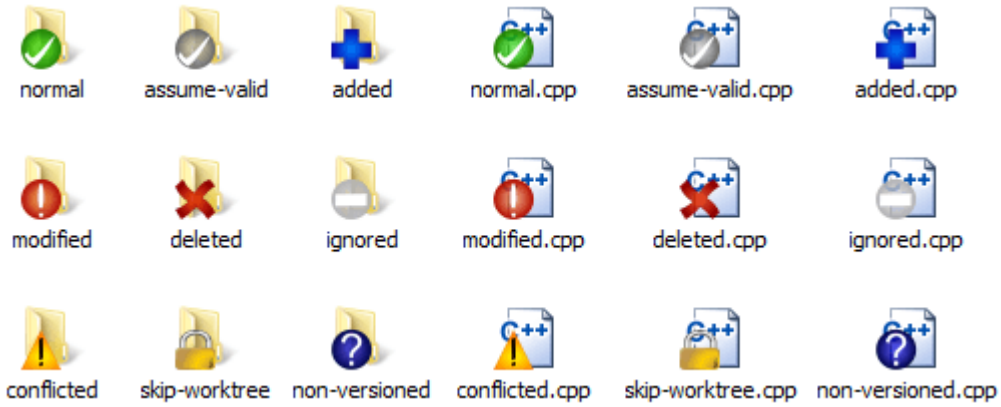


Figure 2.12. Explorer showing icon overlays

Now that you have checked out a working tree you can see your files in the windows explorer with changed icons. This is one of the reasons why TortoiseGit is so popular. TortoiseGit adds a so called overlay icon to each file icon which overlaps the original file icon. Depending on the Git status of the file the overlay icon is different.

A fresh checked out working tree has a green checkmark as overlay. That means the Git status is *normal*.

As soon as you start editing a file, the status changes to *modified* and the icon overlay then changes to a red exclamation mark. That way you can easily see which files were changed since you last updated your working tree and need to be committed.

If during an update a *conflict* occurs then the icon changes to a yellow exclamation mark.

This tells you that this file is marked as "assume-valid". This overlay is optional.

This tells you that this file is marked as "skip-worktree". This overlay is optional.

This icon shows you that some files or folders inside the current folder have been scheduled to be *deleted* from version control.

The plus sign tells you that a file has been scheduled to be *added* to version control.

The bar sign tells you that a file or folder is *ignored* for version control purposes. This overlay is optional.

This icon shows files and folders which are not under version control, but have not been ignored. This overlay is optional.

In fact, you may find that not all of these icons are used on your system. This is because the number of overlays allowed by Windows is very limited and if you are also using an old version of TortoiseCVS or tools with overlay handlers such as SkyDrive, DropBox or GoogleDrive, then there are not enough overlay slots available. TortoiseGit tries to be a “Good Citizen (TM)” and limits its use of overlays to give other apps a chance too.

If you have problems with overlays, please see the online *FAQ* [https://tortoisegit.org/support/faq/#ovlnotshowing].

For a description of how icon overlays correspond to Git status and other technical details, read Section E.1, “Icon Overlays”.

2.6.2. Explorer Properties

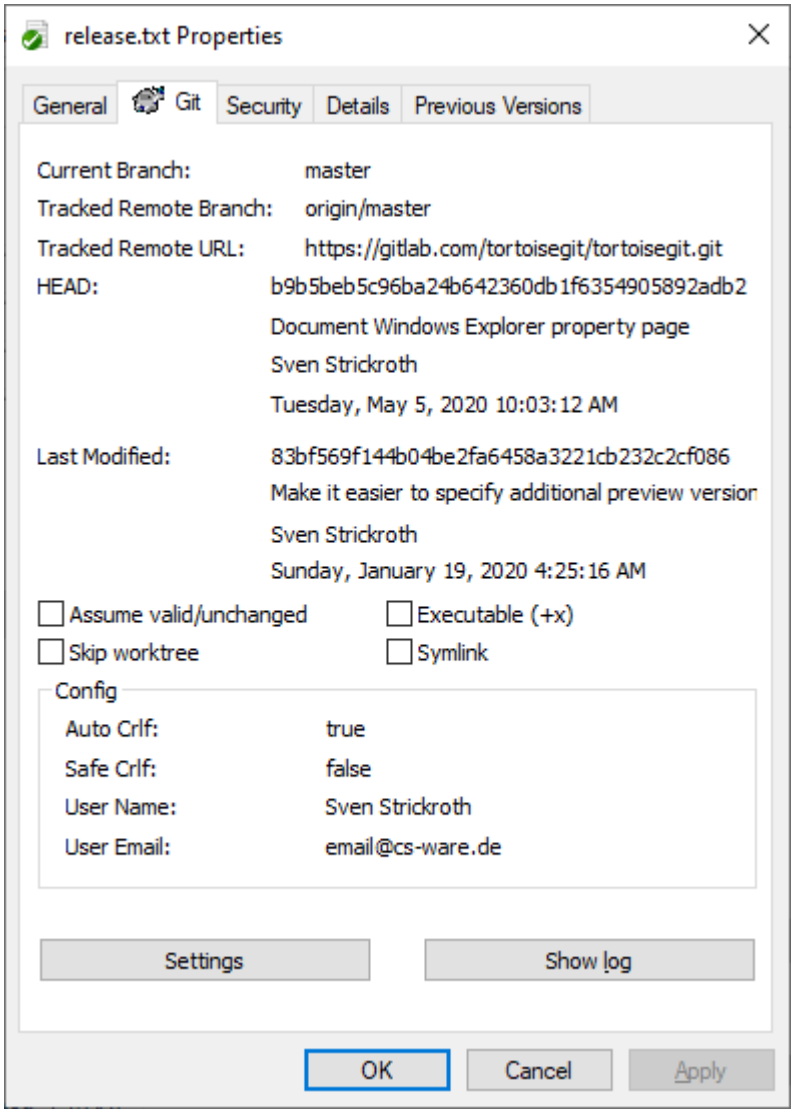


Figure 2.13. Explorer property page, Git tab

TortoiseGit provides a property page in Windows Explorer for files/folders under Git version control. There you can see all relevant information about the selected file/directory. Just select a file or directory and select Windows Menu → properties in the context menu (note: this is the normal properties menu entry the explorer provides, not the one in the TortoiseGit submenu!).

2.6.3. Status

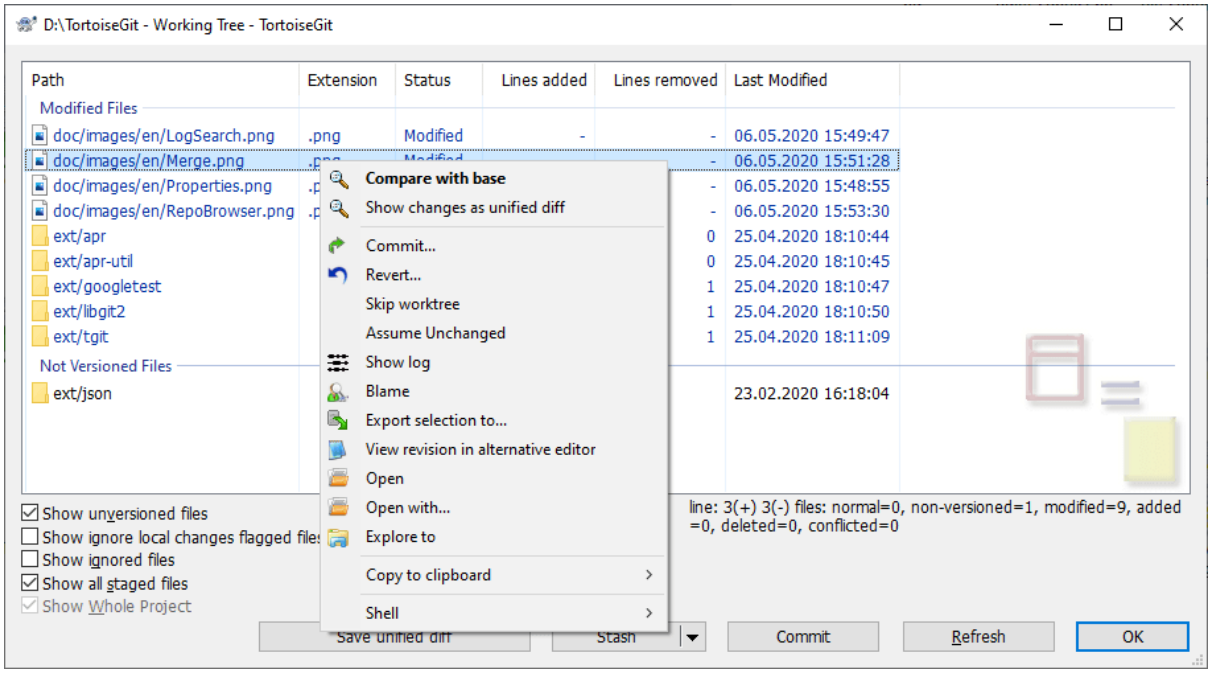


Figure 2.14. Check for Modifications

It's often very useful to know which files you have changed and also which files got changed and committed by others. That's where the command TortoiseGit → Check For Modifications... comes in handy. This dialog will show you every file that has changed in any way in your working tree, as well as any unversioned files you may have.

The dialog uses color coding to highlight the status.

Blue

Locally modified items.

Purple

Added items. Items which have been added with history have a + sign in the Text status column, and a tooltip shows where the item was copied from.

Dark red

Deleted or missing items.

Green

Items modified locally and in the repository. The changes will be merged on update. These *may* produce conflicts on update.

Bright red

Items modified locally and deleted in repository, or modified in repository and deleted locally. These *will* produce conflicts on update.

Black

Unchanged and unversioned items.

This is the default color scheme, but you can customise those colors using the settings dialog. Read Section 2.36.1.6, “TortoiseGit color Settings” for more information.

From the context menu of the dialog you can show a diff of the changes. Check the local changes *you* made using Context Menu → Compare with Base. Check the changes in the repository made by others using Context Menu → Show Differences as Unified Diff.

You can also revert changes in individual files. If you have deleted a file accidentally, it will show up as *Missing* and you can use *Revert* to recover it.

Unversioned and ignored files can be sent to the recycle bin from here using Context Menu → Delete. If you want to delete files permanently (bypassing the recycle bin) hold the **Shift** key while clicking on Delete.

If you want to examine a file in detail, you can drag it from here into another application such as a text editor or IDE.

The columns are customizable. If you right click on any column header you will see a context menu allowing you to select which columns are displayed. You can also change column width by using the drag handle which appears when you move the mouse over a column boundary. These customizations are preserved, so you will see the same headings next time.

If you are working on several unrelated tasks at once, you can also group files together into changelists. Read Section 2.5.2, “Change Lists” for more information.

At the bottom of the dialog you have several options to select which entries to show (such as ignored or untracked/unversioned files). It is also possible to view all files which were marked as "Assume valid" or "Skip worktree" here (using Show ignore local changes flagged files). Resetting those flags (it's also possible to edit this flag using file properties in explorer on the Git tab).

Save unified diff: Generates an unified patch containing the uncommitted changes and opens it with the configured diff viewer. If you want to pass your changes to other people, consider Section 2.32, “Creating and Applying Patches and Pull Requests”.

Stash: Is a drop-down menu for quick access to Git stash (cf. Section 2.25, “Stash Changes”). This feature allows you to put aside uncommitted changes in order to work on an unrelated issue and get the changes back afterwards easily.

Commit: Opens the commit dialog (cf. Section 2.5, “Committing Your Changes To The Repository”) with the very same folder/files the current dialog was opened on. If you want to pre-populate the commit dialog with only a subset of files, select the files and choose Commit... in the context menu.

2.6.4. Viewing Diffs

Often you want to look inside your files, to have a look at what you've changed. You can accomplish this by selecting a file which has changed, and selecting Diff from TortoiseGit's context menu. This starts the external diff-viewer, which will then compare the current file with the pristine copy (BASE revision), which was stored after the last checkout or update.

Tip

Even when not inside a working tree or when you have multiple versions of the file lying around, you can still display diffs:

Select the two files you want to compare in explorer (e.g. using **Ctrl** and the mouse) and choose Diff from TortoiseGit's context menu. The file clicked last (the one with the focus, i.e. the dotted rectangle) will be regarded as the later one.

2.7. Pull and Fetch change

This section talks about how to fetch or pull (i.e., download) changes from another repository. The difference between pull and fetch is:

Fetch just downloads the objects and refs from a remote repository and normally updates the remote tracking branches. Pull, however, will not only download the changes, but also merges them - it is the combination of fetch and merge (cf. Section 2.28, “Merging”). The configured remote tracking branch is selected automatically.

Important

Whenever you merge, it is possible the a file was changed in both branches and that the changes cannot be merged automatically: This is called a "conflict" and needs to be manually resolved. See Section 2.31, “Resolving Conflicts” for more information.

A pull/fetch can be initiated by using TortoiseGit → Pull... or TortoiseGit → Fetch.... Fetching and pulling changes is also possible using the Sync dialog (cf. Section 2.9, “Sync”), however, there you have less options, but the sync dialog allows you to initiate other operations such as pushing and to see diffs and changes.

The fetch and pull dialog will open.

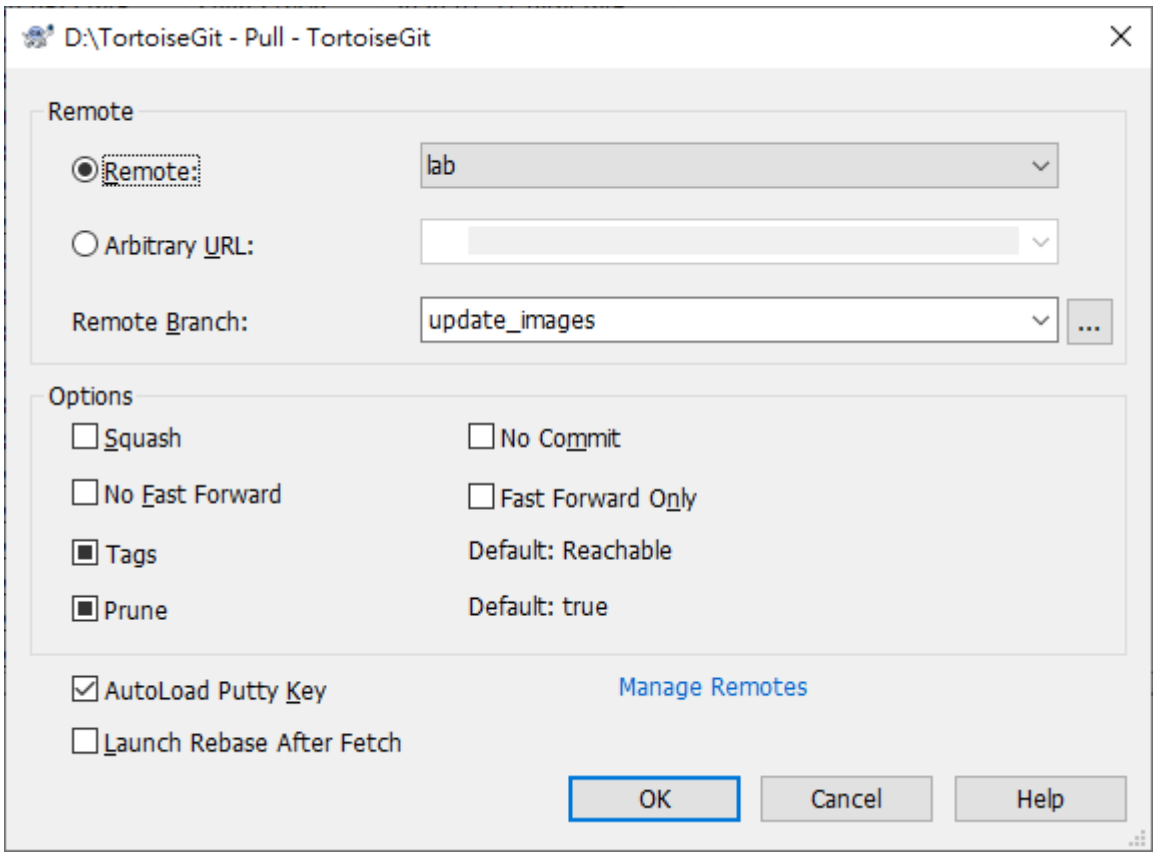


Figure 2.15. Pull dialog

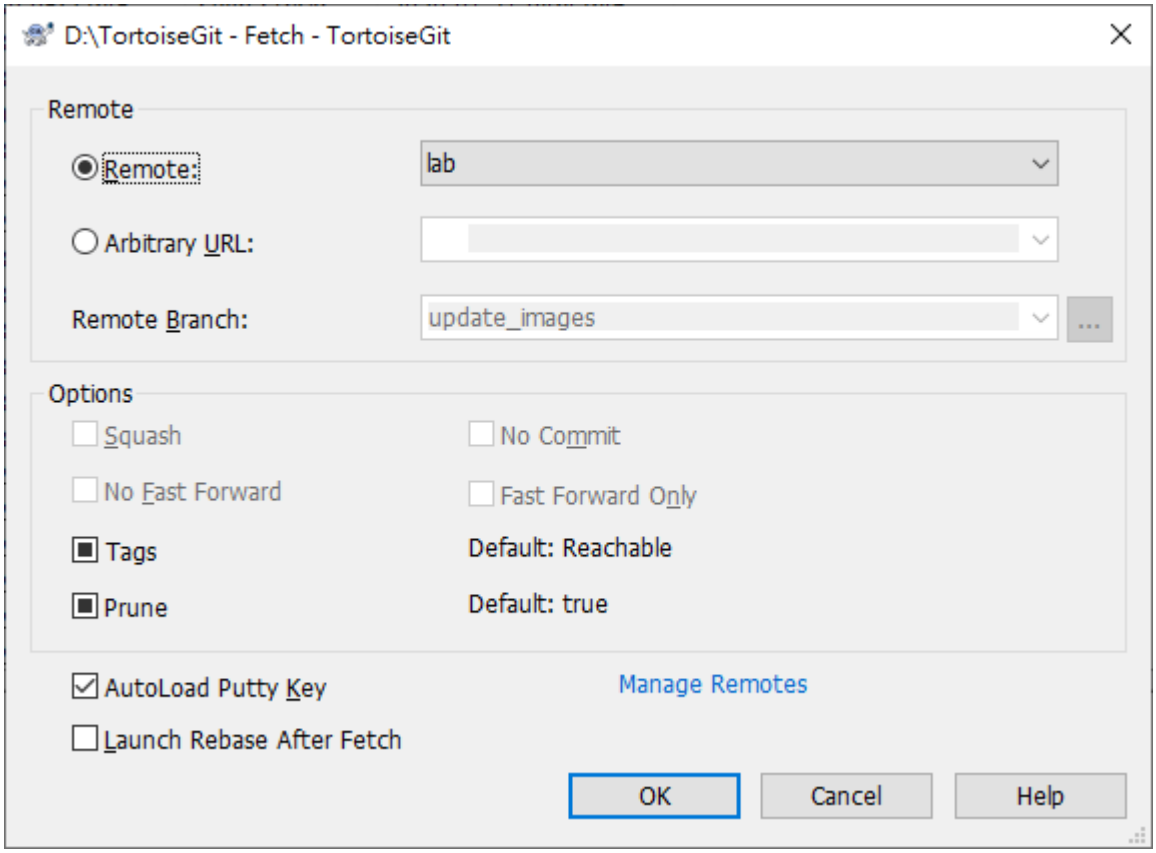


Figure 2.16. Fetch dialog

Remote Choose a configured remote repository (these can be changes using the Manage Remotes label). Instead of the configured repositories, you can also put the URL of another repository into the Arbitrary URL textbox.

If the current active branch has a remote tracked branch set, the remote branch and remote repository are automatically selected. A remote tracked branch can be set using the reference browser (cf. Section 2.11, "Browse All Refs") or using the push dialog (cf. Section 2.8, "Push").

Other: Input Other URL or local directory. You can click ... to browse directory.

If you check the Autoload Putty Key checkbox, a configured Putty key will be automatically loaded using Pageant.

Tags has three states (git 1.9 and later): `Checked`: All tags as well as branches are downloaded (`--tags` is passed to git), `unchecked`: No tags are downloaded (`--no-tags` is passed to git), and third state: use default behavior (based on `remote.<name>.tagopt` setting). Tags has three states (prior to git 1.9): `Checked`: Only all tags are downloaded but no branches are downloaded (`--tags` is passed to git), `unchecked`: No tags are downloaded (`--no-tags` is passed to git), and third state: use default behavior (based on `remote.<name>.tagopt` setting).

Prune has three states: `True` to remove remote-tracking branches which no longer exist on the remote, `false`: not to remove, and third state: use default behavior (based on `remote.<name>.prune` or `fetch.prune` git setting which can be set on Section 2.36.6.3, "Remote").

Tip

You can find more information about PuTTY and using SSH keys at Appendix F, *Tips and tricks for SSH/PuTTY*. There is also explained how you can use several accounts at the same time for a remote.

Conflicts

Although major merge work is done by git automatically while pulling, a conflict may happen during cherry-picking (i.e., a file was modified in your current branch and also in the branch you are pulling), please see Section 2.31, “Resolving Conflicts” on how to resolve conflicts.

Please note, that "REMOTE"/"theirs" in the conflict editor refers to the changes your on the branch you selected for pulling/merging and "LOCAL"/"mine" to your HEAD version in your working tree.

You can find more information at Section G.3.49, “git-fetch(1)” and Section G.3.102, “git-pull(1)”.

2.8. Push

This section talks about how to push (i.e., send) changes to another repository.

In order to perform a push open the push dialog using TortoiseGit → Push... . Pushing changes is also possible using the Sync dialog (cf. Section 2.9, “Sync”), however, there you have less options, but the sync dialog allows you to initiate other operations such as pulling and to see diffs and changes.

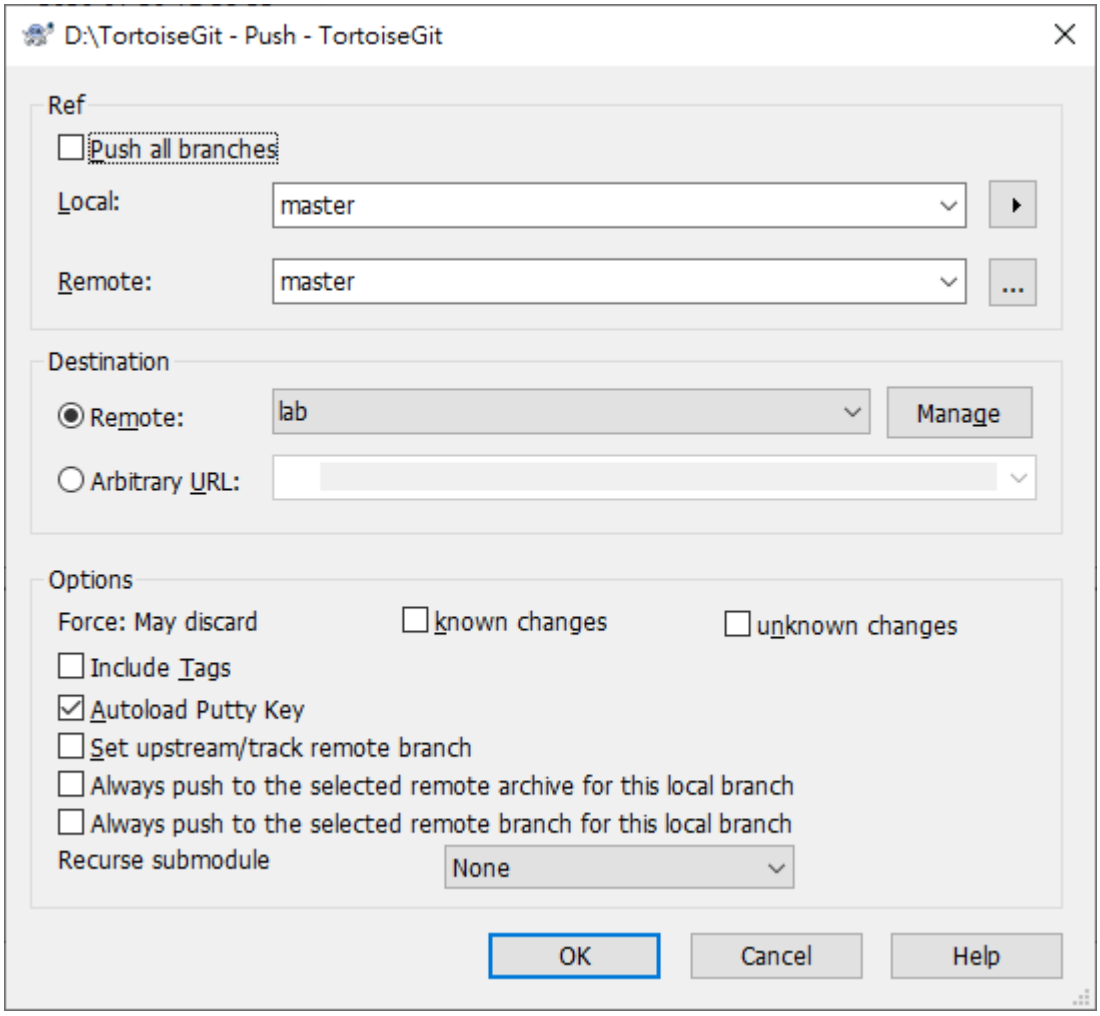


Figure 2.17. Push dialog

2.8.1. Branch

Local: The source branch which will be pushed to the other repository. If the current branch or the selected local branch has a remote tracked branch set, the remote branch and remote repository are automatically selected. A

remote tracked branch can be set using the reference browser (cf. Section 2.11, “Browse All Refs”) or by using Set upstream/track remote branch. This can be overridden in this dialog by using one of the Always push to the selected remote ... options, so that for pushing a different branch is selected automatically as for merging and pulling.

Remote: The remote branch of the other repository.

2.8.2. Destination

Remote: Choose an already configured remote repository.

Arbitrary URL: The URL of a remote repository.

You must push change to a bare repository. Pushing changes to repository which has a working tree can lead to unexpected results.

2.8.3. Options

force with lease This allows remote repository to accept a safer non-fast-forward push. This can cause the remote repository to lose commits; use it with care. This can prevent from losing unknown changes from other people on the remote. It checks if the server branch points to the same commit as the remote-tracking branch (known changes). If yes, a force push will be performed. Otherwise it will be rejected. Since git does not have remote-tracking tags, tags cannot be overwritten using this option. This passes `--force-with-lease` option of `git push` command.

force This allows remote repository to accept an unsafe non-fast-forward push. This can cause the remote repository to lose commits; use it with care. This does not check any server commits, so it is possible to lose unknown changes on the remote. Use this option with Include Tags to overwrite tags. This passes the traditional `--force` option of `git push` command.

Include Tags Also push tags to remote repository.

Autoload Putty Key

Tip

You can find more information about PuTTY and using SSH keys at Appendix F, *Tips and tricks for SSH/PuTTY*. There is also explained how you can use several accounts at the same time for a remote.

Set upstream/track remote branch: After a successful push, the tracking relationship will be set between the pushed local branch and its remote tracking branch. This will select the remote branch automatically for pulling/pushing and merging.

Always push to the selected remote archive for this local branch

Always push to the selected remote branch for this local branch

Recurse submodule `None`: No checking. `Check`: Checks if the bounded commits of all submodules are present on the remote repositories. If any of the submodules are not pushed, the super project push will fail. `On-demand`: Checks if the bounded commits of all submodules are present on the remote repositories. If the submodules are not pushed yet, it will try to push them.

You can find more information at Section G.3.103, “git-push(1)”.

2.9. Sync

The Sync Dialog provides an interface for all operations related with remote repositories in one dialog. This includes push, pull, fetch, remote update, submodule update, send patch... However, the sync dialog provides less options as the regarding dialogs (cf. Section 2.7, “Pull and Fetch change” and Section 2.8, “Push”).

The sync dialog can be opened using Sync... .

The Sync Dialog will show.

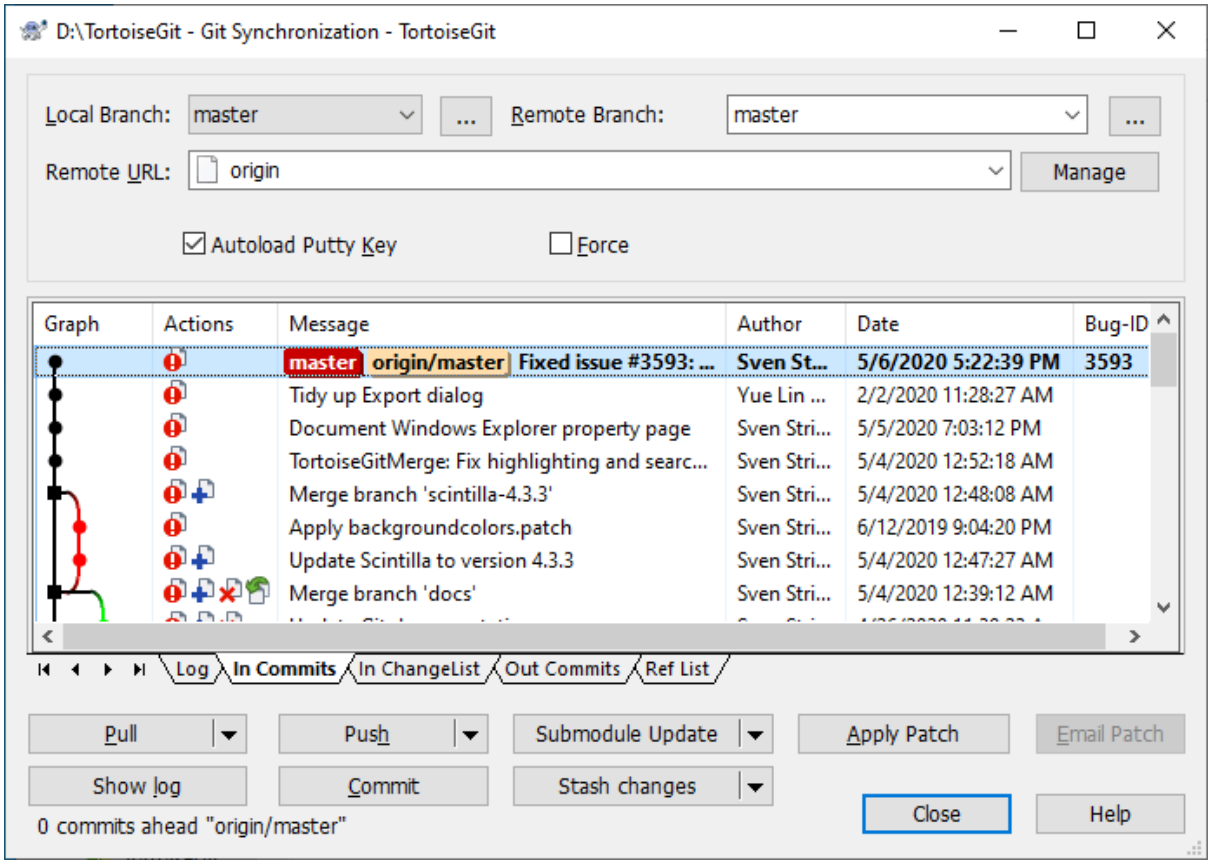


Figure 2.18. Sync dialog

2.9.1. Branch

Local Branch: The source branch which will push/pull to/from other repository. If the current branch or the selected local branch has a remote tracked branch set, the remote branch and remote repository are automatically selected. A remote tracked branch can be set using the reference browser (cf. Section 2.11, “Browse All Refs”) or using the push dialog (cf. Section 2.8, “Push”).

Remote Branch: The remote branch of a remote repository.

2.9.2. Destination

Remote URL: Choose remote repository or input remote repository URL.

Manage Add new remote name.

2.9.3. Options

Force Force Overwrite Existing Branch(May discard changes)

Autoload putty key Autoload putty key when push or pull

Tip

When you press the **shift**-key while selecting fetch, pull, fetch, stash changes or submodule update/sync you can open the separate dialog boxen which allows you to control more parameters for the specific operations.

2.10. Daemon

Sometimes you want to quickly share you local repository to others without pushing to a remote git repository. That's when you need to use TortoiseGit → Daemon....

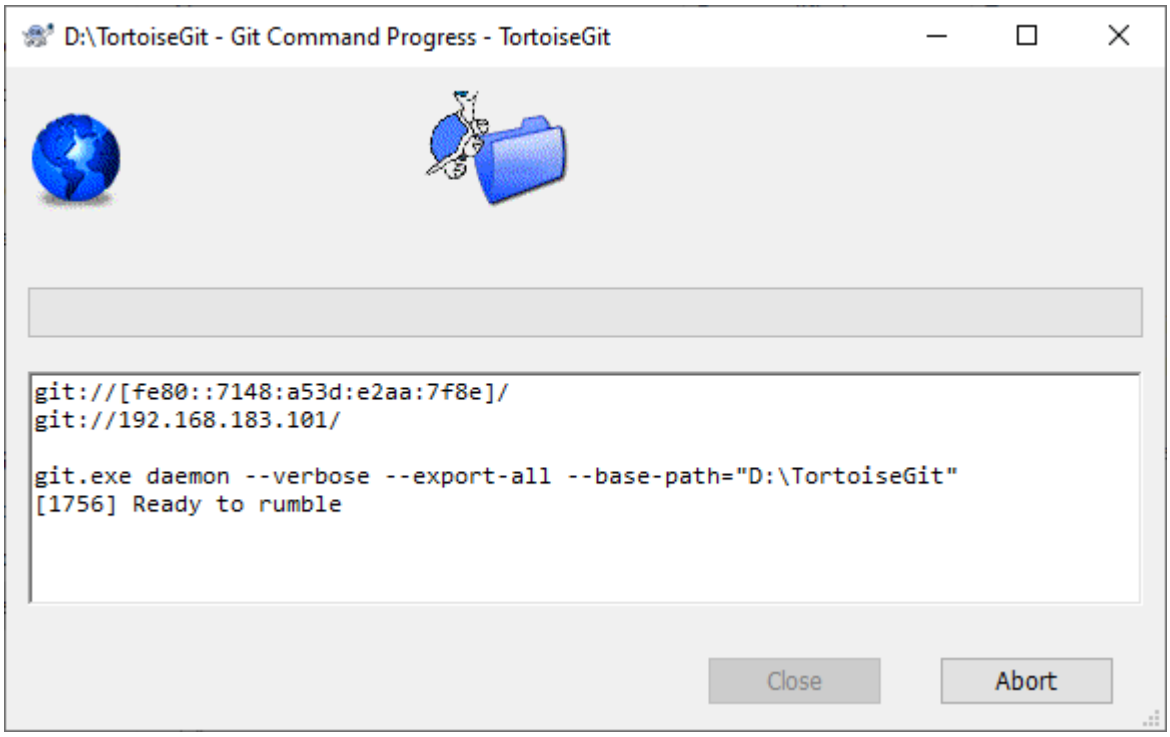


Figure 2.19. A running daemon dialog

This command runs Git Daemon that serves Git protocol at port 9418 (git://hostname/).

Caution

The selected repository is exported for read/write access without further authentication.

Important

Your host might only be accessible within your local network and you might need to adjust your firewall.

You can find more information at Section G.3.38, “git-daemon(1)”.

2.11. Browse All Refs

This section talks about the reference browser, which allows you to view and work with all refs (tags, branches, remote branches, stash and so on). It can be opened using TortoiseGit → Browse Reference....

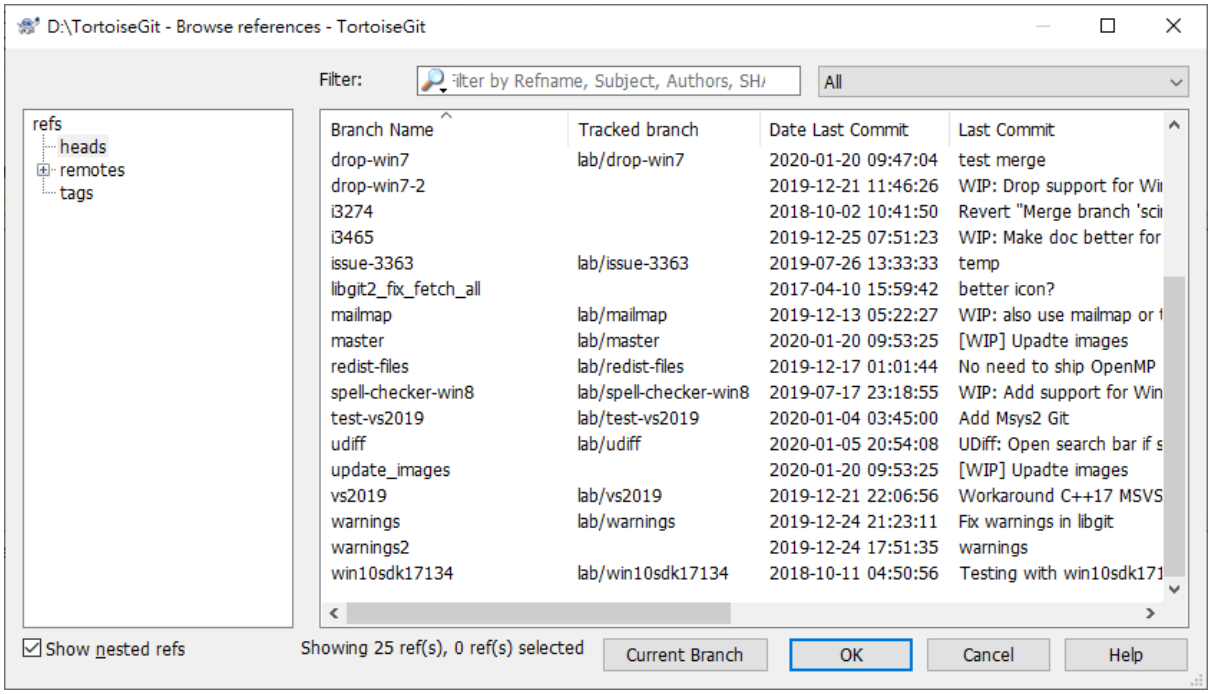


Figure 2.20. Browse References Dialog dialog

The left panel displays the ref "types" in a tree such as tags, heads (local branches) and so on.

Right panel shows all refs for a selected type (recursively if not disabled using Show nested refs) including the latest commit, description and (for local branches) their remote tracked branch.

At the top there is an edit bar which can be used to filter the displayed refs in the right panel. The search syntax is similar to the one available in the Log dialog (cf. Section 2.14.6, “Filtering Log Messages”).

On both panels there is a powerful context menu which provides further options such as deleting/renaming refs, configuring the remote tracked branch (for local branches) and deleting tags for a remote (on the left panel when a remote is selected). If exactly two refs are selected it is possible to compare them or open the log for all commits which are on both branches (Show log of branch1...branch2) or just on one of the two (Show log of branch1..branch2).

In order to delete remote tags, use the context menu on a remote on the left side and select Delete remote tags.... Then the following dialog will come up. There you can delete multiple remote tags at once.

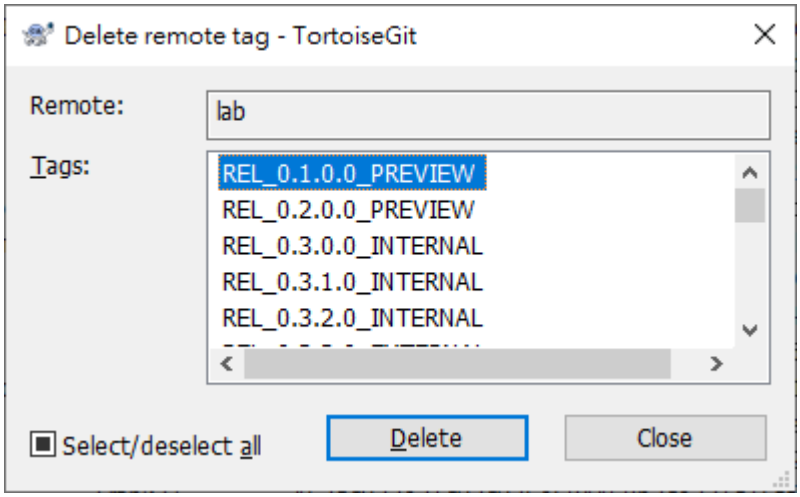


Figure 2.21. Delete remote tags dialog

2.12. Submodules

When you want to embed foreign repositories into a working tree/git repository, this is called a submodule. Here using the TortoiseGit → Submodules Add option a foreign repository can be embedded into a dedicated sub directory of the source tree. When selecting this option, a dialog pops up:

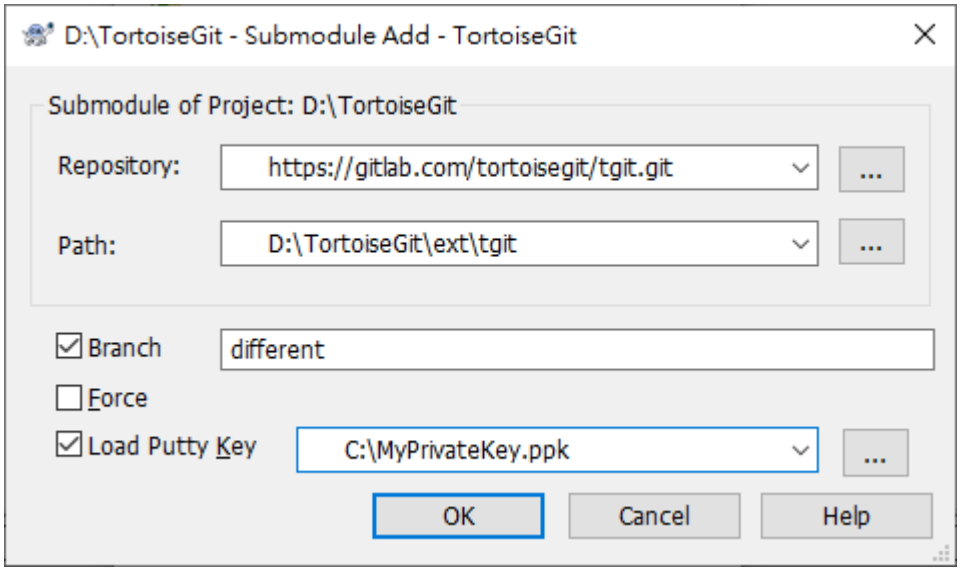


Figure 2.22. The add submodule dialog

Here you can enter the location/URL of the Repository you want to embed into the directory Path. Path can be entered as a relative path within the active source tree, but can also be an absolute path (pointing to the active source tree). The folder should be empty or non existent. If you don't want to integrate the HEAD of the Repository, you can enter a different Branch. By pressing OK, the entered Repository is cloned and integrated into the current source tree.

If a working tree contains submodules, two new context menu entries are available:

Figure 2.23. Submodule context menu entries

Submodule Update:

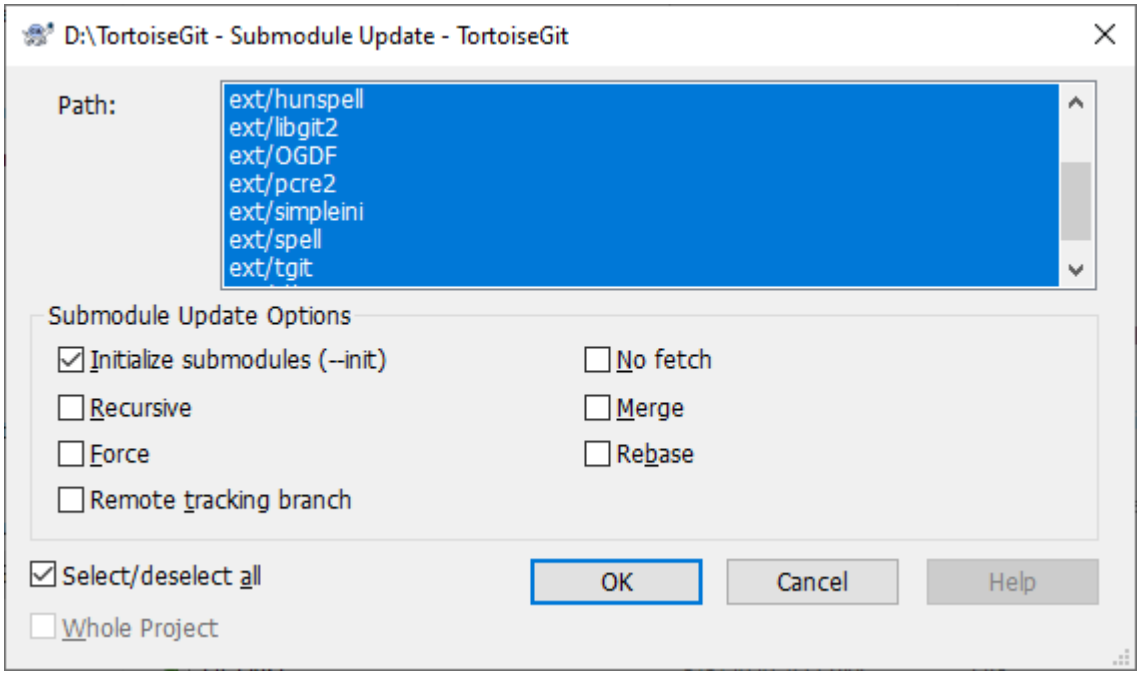


Figure 2.24. The update submodule dialog

Initialize the submodules and/or update the registered submodules, i.e. clone missing submodules and checkout the commit specified in the index of the containing repository.

Submodule Sync: Synchronizes submodules' remote URL configuration setting to the value specified in `.gitmodules`. This is useful when submodule URLs change upstream and you need to update your local repositories accordingly.

Also if a working tree contains submodules, Section 2.4, “Checking Out A Working Tree (Switch to commit)” and Section 2.24, “Reset” contain a button for updating submodules:

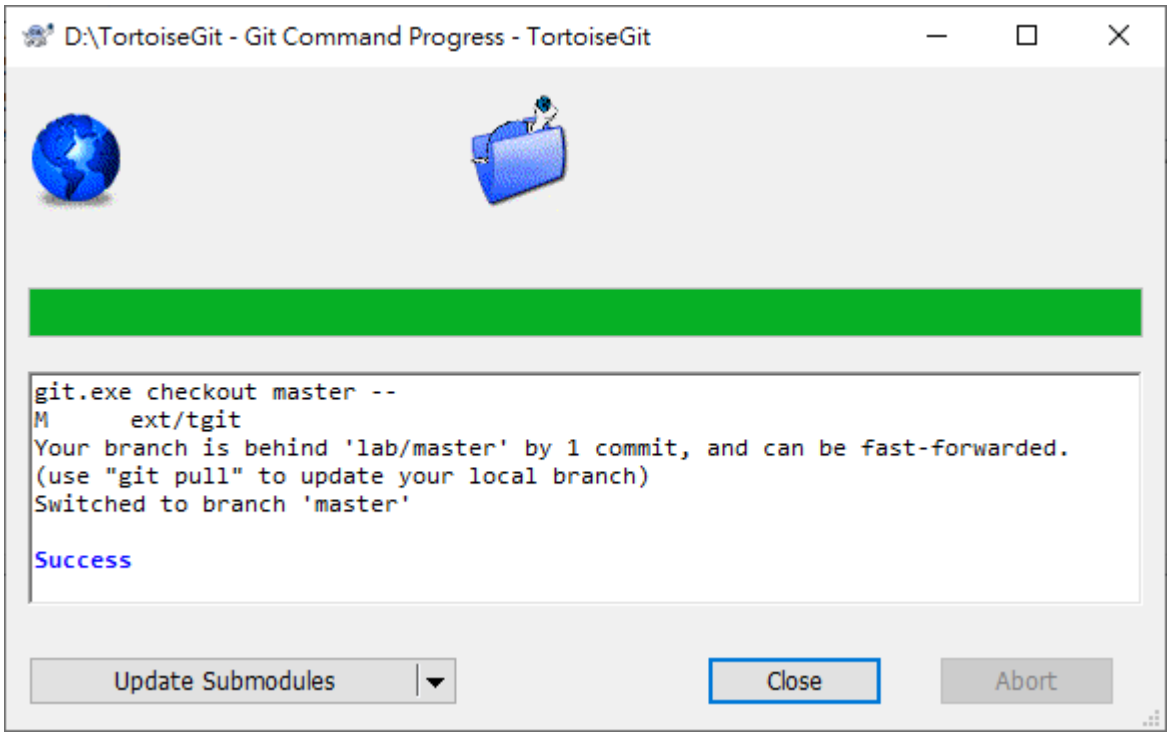


Figure 2.25. Button for updating submodules in progress dialog

You can find more information at Section G.3.141, “git-submodule(1)”.

2.13. Change Lists

In an ideal world, you only ever work on one thing at a time, and your working tree contains only one set of logical changes. OK, back to reality. It often happens that you have to work on several unrelated tasks at once, and when you look in the commit dialog, all the changes are mixed in together. The *changelist* feature helps you group files together, making it easier to see what you are doing. Of course this can only work if the changes do not overlap. If two different tasks affect the same file, there is no way to separate the changes.

Tip

There is a similar functionality in Git called stashing (Section 2.25, “Stash Changes”).

Generally you should consider to create a new branch as Git commits are just local, you're not messing with everybody's repository, but just your own.

You can see changelists in several places, but the most important ones are the commit dialog and the check-for-modifications dialog. Let's start in the check-for-modifications dialog after you have worked on several features and many files. When you first open the dialog, all the changed files are listed together. Suppose you now want to organize things and group those files according to feature.

Select one or more files and use Context Menu → Move to changelist to add an item to a changelist. Initially there will be no changelists, so the first time you do this you will create a new changelist. Give it name which describes what you are using it for, and click OK. The dialog will now change to show groups of items.

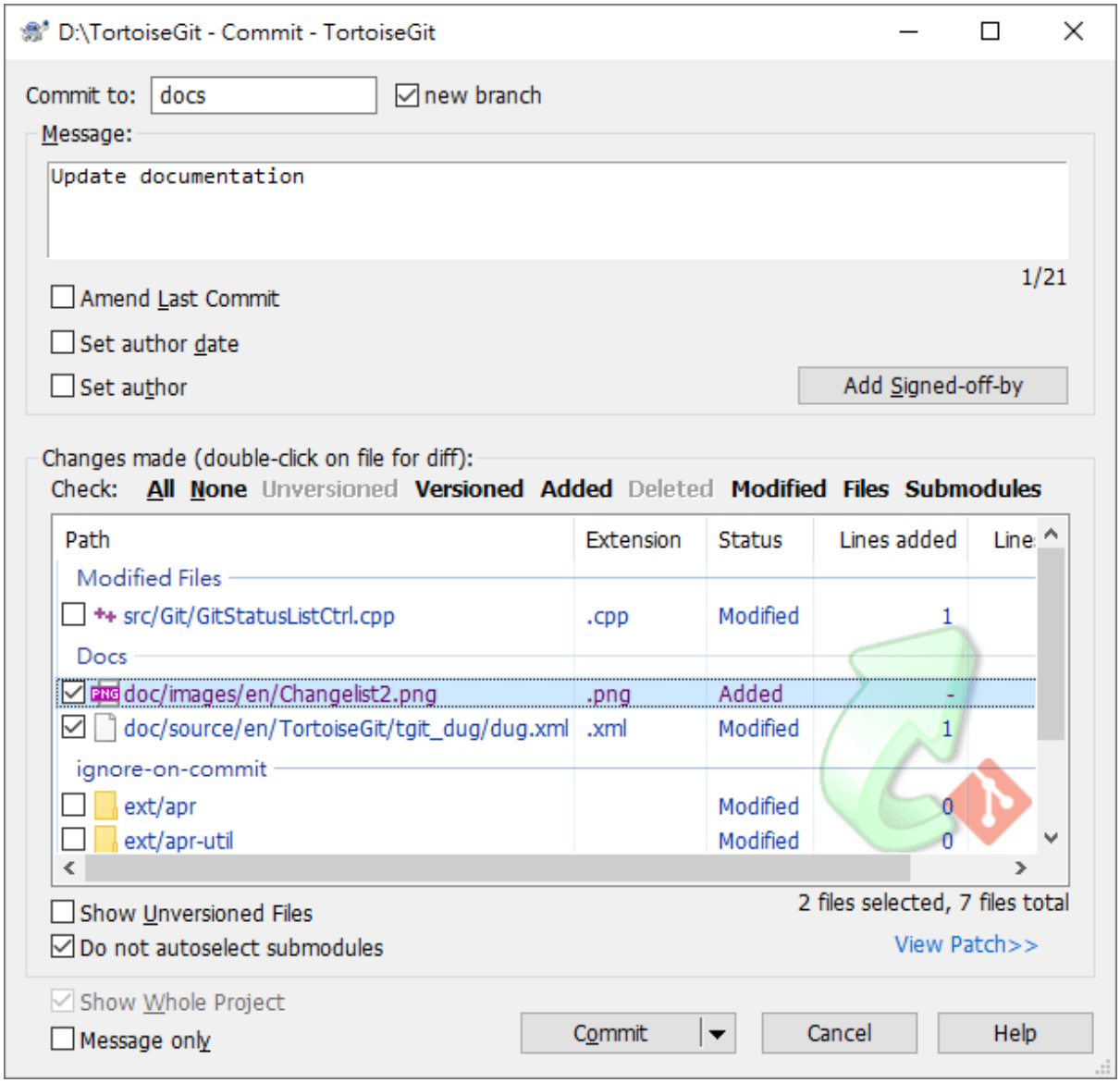


Figure 2.26. Commit dialog with Changelists

In the commit dialog you can see those same files, grouped by changelist. Apart from giving an immediate visual indication of groupings, you can also use the group headings to select which files to commit.

Click on the group header to select all entries, then check one of the selected entries to check all.

TortoiseGit reserves one changelist name for its own use, namely `ignore-on-commit`. This is used to mark versioned files which you almost never want to commit even though they have local changes. This feature is described in Section 2.5.3, “Excluding Items from the Commit List”.

When you commit files belonging to a changelist then normally you would expect that the changelist membership is no longer needed. So by default, files are removed from changelists automatically on commit.

Tip

Changelists are purely a local TortoiseGit client feature. Creating and removing changelists will not affect the repository, nor anyone else's working copy. They are simply a convenient way for you to organize your files.

2.14. Log Dialog

For every change you make and commit, you should provide a log message for that change. That way you can later find out what changes you made and why, and you have a detailed log for your development process.

The Log Dialog retrieves all those log messages and shows them to you. The display is divided into 3 panes.

- The top pane shows a list of revisions where changes to the file/folder have been committed. This summary includes the date and time, the person who committed the revision and the start of the log message.

  The line shown in bold indicates the HEAD commit and the entry "Working tree changes" is a virtual entry representing the current (uncommitted) state of your working tree.

- The middle pane shows the full log message for the selected revision.

- The bottom pane shows a list of all files and folders that were changed as part of the selected revision.

But it does much more than that - it provides context menu commands which you can use to get even more information about the project history.

2.14.1. Invoking the Revision Log Dialog

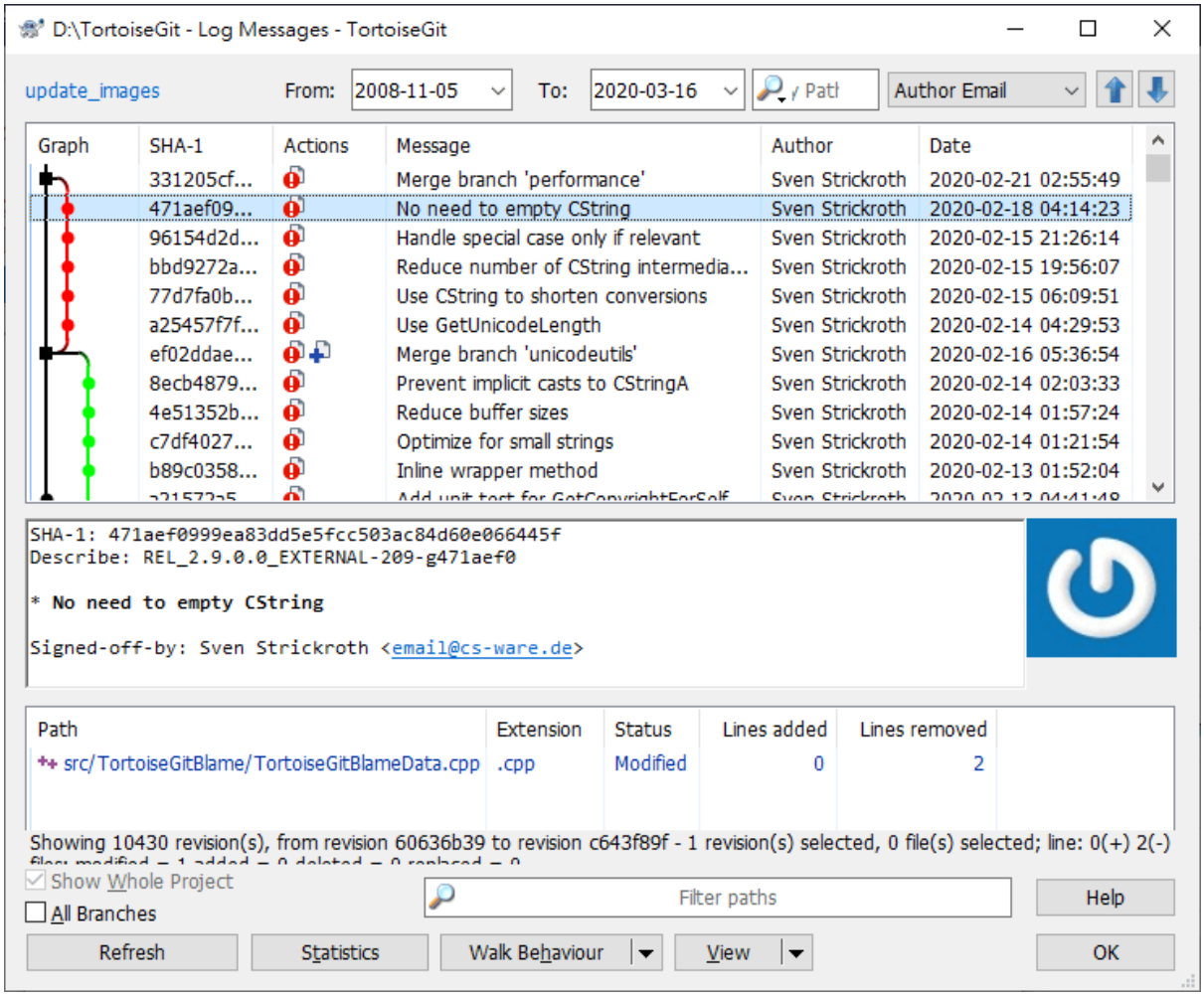


Figure 2.27. The Revision Log Dialog

There are several places from where you can show the Log dialog:

- From the explorer context menu using TortoiseGit → Show log....

- From various TortoiseGit dialogs where you can select a commit (oftentimes using a ... button).
- From various TortoiseGit dialogs where commit entries or files are shown using the context menu.

2.14.2. Revision Log Actions

The top pane has an Actions column containing icons that summarize what has been done in that revision. There are four different icons, each shown in its own column.

If a revision modified a file or directory, the *modified* icon is shown in the first column.

If a revision added a file or directory, the *added* icon is shown in the second column.

If a revision deleted a file or directory, the *deleted* icon is shown in the third column.

If a revision replaced(rename) a file, the *replaced* icon is shown in the fourth column.

2.14.3. The revision graph

The top pane has a Graph column with a line showing the commits, merges and branches. There are two types of shapes.

The circles indicate normal commits w/o any branching.

The squares indicate merges and branchings. In the first case these are merge commits (i.e., a commit with more than one parent) and in the latter case these are also normal commits. A square was chosen here to indicate that a branch goes off.

The colors of the lines (and of the shapes) are there to make the graph better readable (i.e., to see to which branch they belong). The colors are configurable (cf. Section 2.36.1.8, “TortoiseGit color Settings 3”).

2.14.4. Commit messages and branch/tag indicators

The top pane also has a message column. In this column the subjects of the commits are shown. Commits which have an associated reference are decorated with a label.

Tags are by default displayed as yellow rectangles. In Git there are two tag types: normal tags and annotated tags. The annotated ones have an apex at the right side.

Normally branches are displayed as normal rectangles. The active branch is displayed as a dark red rectangle (by default), the green ones are local branches and the peach ones are remote branches.

The boxes with rounded corners for local branches indicate that it has an associated remote tracking. The boxes with rounded corners for remote tracking branches are used to indicate which of (possible several remote branches) is the corresponding remote tracking branch (e.g., master and origin/master).

The stash has a dark gray rectangle.

There are also rectangles to indicate the bad versions (light red) on bisecting, blue for known good and gray for skip.

Depending on what branch/tag label you open the context menu, it is optimized for that ref it was opened on (e.g., a local branch offers directly to switch to it and to push it whereas a remote branch has no push option).

The colors are configurable (cf. Section 2.36.1.7, “TortoiseGit color Settings 2”). Also, the labels can be configured to be a the right side of the commit message or the displayed text to contain the whole of the commit message (cf. Section 2.36.1.3, “TortoiseGit Dialog Settings”).

2.14.5. Getting Additional Information

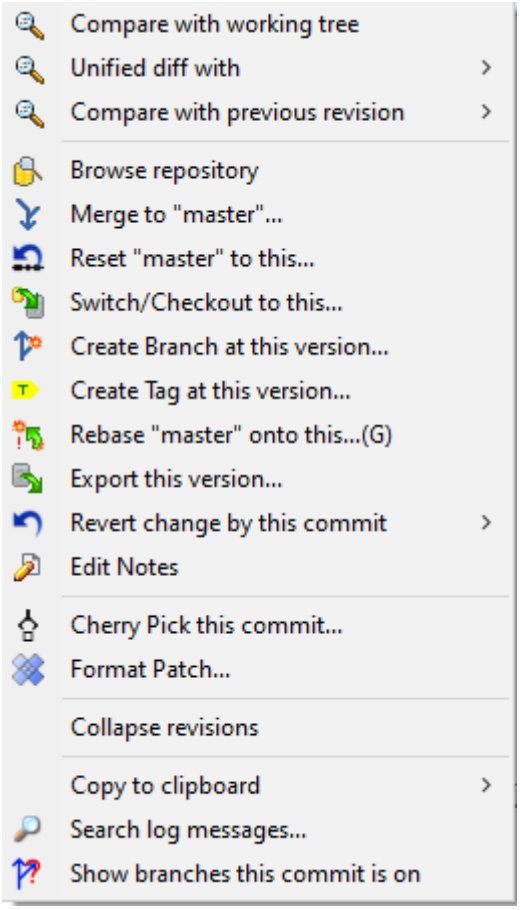


Figure 2.28. The Revision Log Dialog Top Pane with Context Menu

The top pane of the Log dialog has a context menu that allows you to access much more information. You can press the **Shift** key to see the extended menu with some more options (e.g., pushing and opening the log from every commit).

Compare with working tree

Compare the selected revision with your working tree. The default Diff-Tool is TortoiseGitMerge which is supplied with TortoiseGit. If the log dialog is for a folder, this will show you a list of changed files, and allow you to review the changes made to each file individually.

Show changes as unified diff

View the changes made in the selected revision as a Unified-Diff file (GNU patch format). This shows only the differences with a few lines of context. It is harder to read than a visual file compare, but will show all file changes together in a compact format.

Compare with previous revision

Compare the selected revision with the previous revision. This works in a similar manner to comparing with your working tree. For folders this option will first show the changed files dialog allowing you to select files to compare.

Browse repository

Open the repository browser to examine the selected file or folder in the repository as it was at the selected revision (cf. Section 2.17, “The Repository Browser”).

Reset (current branch) to this

Resets the HEAD to the selected commit (cf. Section 2.24, “Reset”).

Switch / Checkout to revision

Update your working tree to the selected revision. Useful if you want to have your working tree reflect a time in the past, or if there have been further commits to the repository and you want to update your working tree one step at a time.

Create branch from revision

Create a branch based on the selected revision (cf. Section 2.27, “Branching/Tagging”).

Create tag from revision

Create a tag on a selected revision (cf. Section 2.27, “Branching/Tagging”).

Rebase (current branch) to this

Rebase current branch on top of the selected commit (cf. Section 2.30, “Rebase”).

Export this version...

Export the selected revision to an archive file such as zip. This brings up a dialog for you to confirm the revision, and select a location for the export (cf. Section 2.34, “Exporting a Git Working Tree”).

Revert change by this commit

Revert changes from which were made in the selected revision. All changes are integrated into your working tree. You may choose to commit immediately or edit and commit later. To abandon the reverted changes, perform a hard reset.

Edit notes

Edit notes of the selected commit.

Collapse revisions

Do not show parent commits up to a merge or branch point or a commit marked with a branch or tag. This is similar to Walk Behavior → Compressed Graph, but only for selected revisions.

Collapsed items are displayed in the graph with hollow circles and squares. In the following screenshot the commits of the merged branch in which issue 3764 was fixed and also all commits starting from the libgit2 update commit until the next merge commit were collapsed.

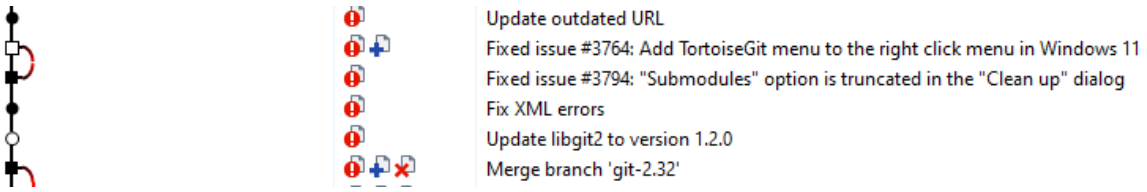


Figure 2.29. Collapsed revisions

Expand hidden revisions

This is shown for collapsed items instead of Collapse revisions and reverts the action.

Cherry Pick this commit

Cherry Pick this commit on top of HEAD (cf. Section 2.29, “Cherry picking”).

Bisect start

Start bisection. Find by binary search the change that introduced a bug (cf. Section 2.26, “Bisect”).

Format Patch...

Create Patches from this commit.

Copy SHA-1 to clipboard

Copy the commit hash of the selected revision to the clipboard.

Copy to clipboard

Copy the log details of the selected revisions to the clipboard. This will copy the revision number, author, date, log message and the list of changed items for each revision.

Copy log message to clipboard

Copy the log message of the selected revision to the clipboard.

Search log messages...

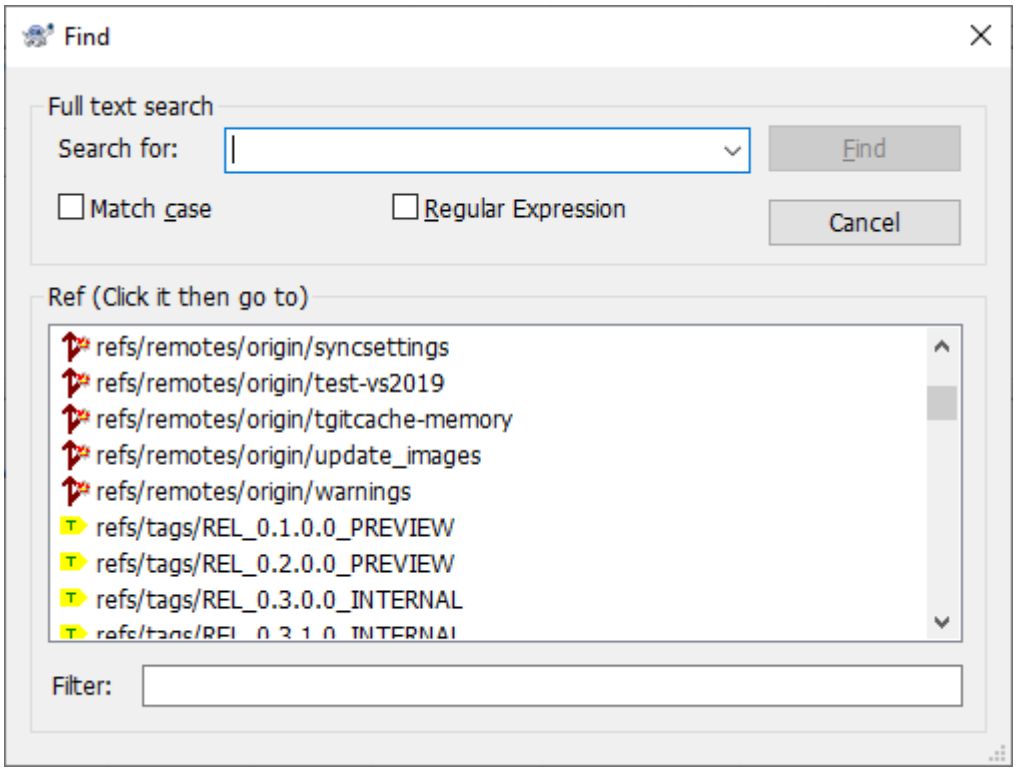


Figure 2.30. The Search Log Messages Dialog

Search log messages for the text you enter. This searches the log messages that you entered and also the action summaries created by Git (shown in the bottom pane).

Tip

This allows you to easily search for refs (tags and branches).

If you press **SHIFT** while clicking on a ref or on Find you can navigate to the commit w/o selecting it.

The Search Log Messages dialog is complementary to the Log Filter which is implemented using the controls at the top of the Log dialog (cf. Section 2.14.6, “Filtering Log Messages”).

Shows branches this commit is on

Shows the branches that the select commit belongs to. It shows both local and remote branches.

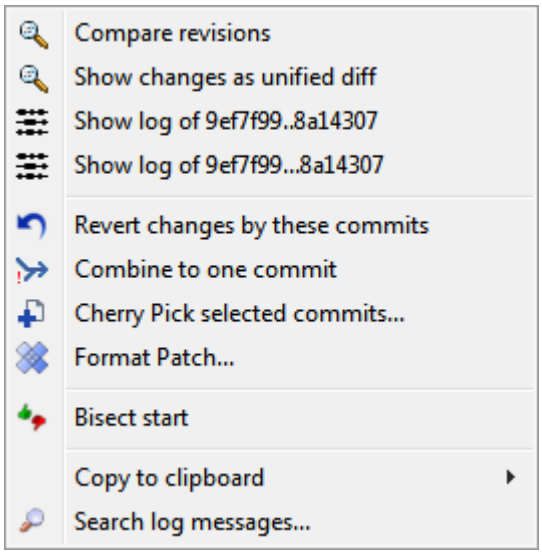


Figure 2.31. Top Pane Context Menu for 2 Selected Revisions

If you select two revisions at once (using the usual **Ctrl**-modifier), the context menu changes and gives you fewer options:

Compare revisions

Compare the two selected revisions using a visual difference tool. The default Diff-Tool is TortoiseGitMerge which is supplied with TortoiseGit.

Show differences as unified diff

View the differences between the two selected revisions as a Unified-Diff file. This works for files and folders.

Revert changes by these commits

Revert changes from which were made in the selected revisions. All changes are integrated into your working tree. You may choose to commit immediately or edit and commit later. To abandon the reverted changes, perform a hard reset.

Combine to one commit

Combine continuous commits to one commit.

Cherry Pick selected commits

Cherry Pick chosen Commits on top of current HEAD (cf. Section 2.29, “Cherry picking”).

Format Patch...

Create patches between two chosen commits.

Copy SHA-1 to clipboard

Copy the commit hashes of the selected revisions to the clipboard, delimited by CRLF.

Copy to clipboard

Copy log messages to clipboard as described above.

Copy log messages to clipboard

Copy the log messages of the selected revisions to the clipboard. This will copy the log message for each revision. This facilitates the preparation of release notes.

Search log messages...

Search log messages as described above.

If you select two or more revisions (using the usual **Ctrl** or **Shift** modifiers), You can combine select commits to one commit. And cherry pick these commits to current branch.

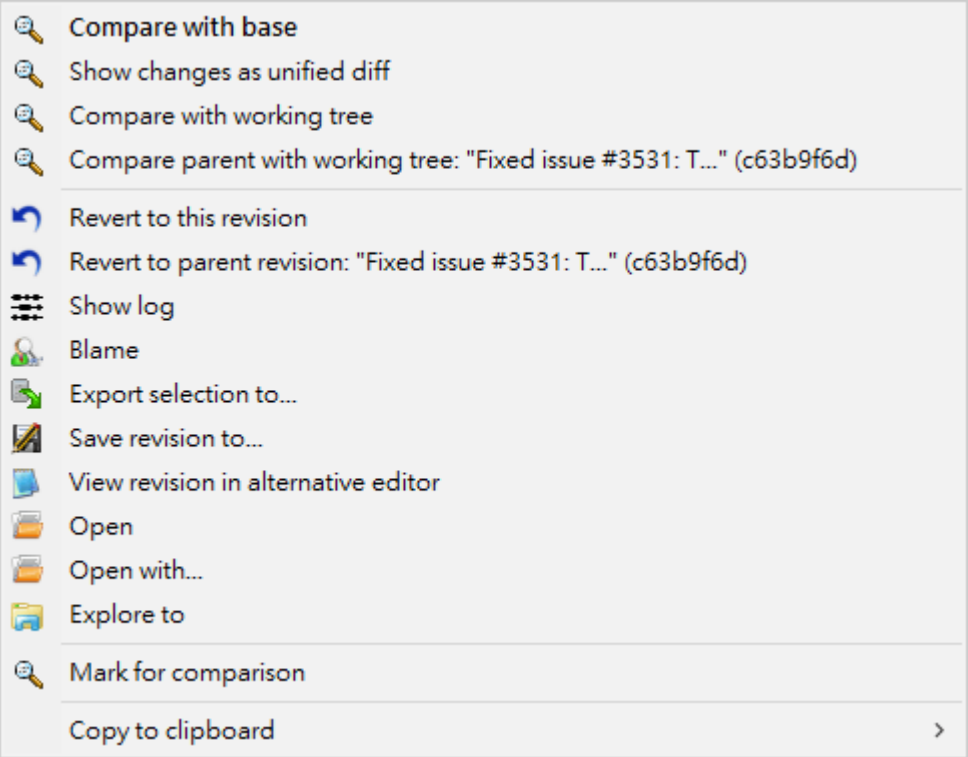


Figure 2.32. The Log Dialog Bottom Pane with Context Menu

The bottom pane of the Log dialog also has a context menu that allows you to

Compare with base

Compare chosen file with base version.

Show as unified diff

Show file changes in unified diff format. This context menu is only available for files shown as *modified*.

Compare with working tree

Compare chosen file with working tree.

Revert changes to this revision

Revert chosen files to the state of this revision.

Revert changes to parent revision

Revert chosen files to the state before this revision.

Show log

Show the revision log for the selected single file.

Blame...

Opens the Blame dialog, allowing you to blame up to the selected revision (cf. Section 2.33, “Who Changed Which Line?”).

Save revision to...

Save the selected revision to a file so you have an older version of that file.

Export selection to...

Saves the selected files to a target directory. Compared to "Save revision to..." this preserves the directory structure.

View revision in alternative editor

Show chosen file with an alternative editor such as notepad2 with chosen commit.

Open/Open with...

Open the selected file, either with the default viewer for that file type, or with a program you choose.

Explore to

Open directory of file with Explore.

Copy paths to clipboard

Copy paths to clipboard

Copy all information to clipboard

Copy all information to clipboard, include version info.

Tip

You may notice that sometimes we refer to changes and other times to differences. What's the difference?

2.14.6. Filtering Log Messages

If you want to restrict the log messages to show only those you are interested in rather than scrolling through a list of hundreds, you can use the filter controls at the top of the Log Dialog.

The first element is the branch/revision filter. Clicking on it opens the Reference Browser (see Section 2.11, “Browse All Refs”). There you can select single or multiple references. If you select exactly two references, you can choose how to combine them (showing especially both A and B "A B"; showing differences "A...B" or all commits between A and B "A..B"). This filter element also contains a special context menu. Here shortcuts for "HEAD", "FETCH_HEAD", "All" and "All local branches" are available. Also, the last manual selected filters are included there.

The start and end date controls allow you to restrict the output to a known date range. The search box allows you to show only messages which contain a particular phrase. A default limitation for From or the number of log entries displayed can be configured in the settings dialog on the Dialogs 1 page (cf. Section 2.36.1.3, “TortoiseGit Dialog Settings”). When a default limitation is active, the From has a blueish background. By opening the context menu on the date picker, you can configure the default limit or disable the default limit for the current dialogue.

Click on the search icon (the magnifier glass) to select which information you want to search in, to choose *regex* mode, and to toggle case sensitivity. Deselecting "Paths" can significantly speed up the filter.

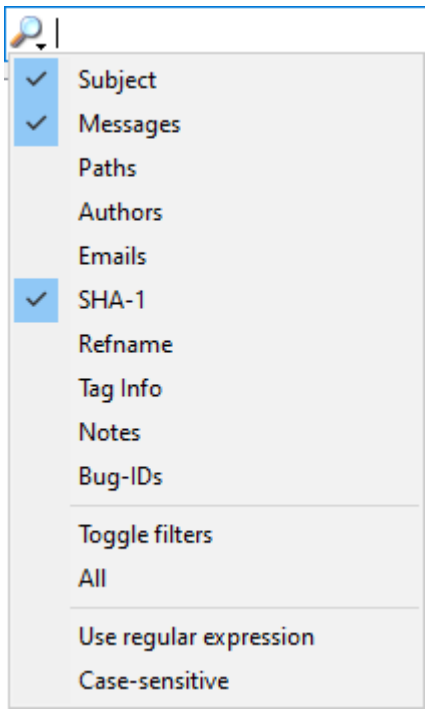


Figure 2.33. Option Menu for the Log Filter

Normally you will only need a simple sub-string search, but if you need to more flexible search terms, you can use regular expressions. If you hover the mouse over the box, a tooltip will give hints on how to use the regex functions. To invert the results for the entire search expression, start the string with an exclamation mark ('!'). You can also find online documentation and a tutorial at *https://www.regular-expressions.info/* [https://www.regular-expressions.info/]. The filter works by checking whether your filter string matches the log entries, and then only those entries which *match* the filter string are shown.

Simple sub-string search works in a manner similar to a search engine. Strings to search for are separated by spaces, and all strings must match. You can use a leading `-` to specify that a particular sub-string is not found (invert matching for that term), and you can use `!` at the start of the expression to invert matching for the entire expression. You can use a leading `+` to specify that a sub-string should be included, even if previously excluded with a `-`. Note that the order of inclusion/exclusion is significant here. You can use quotation marks to surround a string which must contain spaces, and if you want to search for a literal quotation mark you can use two quotation marks together as a self-escaping sequence. Note that the backslash character is *not* used as an escape character and has no special significance in simple sub-string searches. Examples will make this easier:

```
Alice Bob -Eve
```

searches for strings containing both Alice and Bob but not Eve

```
Alice -Bob +Eve
```

searches for strings containing both Alice but not Bob, or strings which contain Eve.

```
-Case +SpecialCase
```

searches for strings which do not contain Case, but still include strings which contain SpecialCase.

```
!Alice Bob
```

searches for strings which do not contain both Alice and Bob

```
!-Alice -Bob
```

do you remember De Morgan's theorem? NOT(NOT Alice AND NOT Bob) reduces to (Alice OR Bob).

```
"Alice and Bob"
```

searches for the literal expression “Alice and Bob”

```
""
```

searches for a double-quote anywhere in the text

```
"Alice says ""hi"" to Bob"
```

searches for the literal expression “Alice says "hi" to Bob”.

You can also filter the path names in the bottom pane using the View → Hide unrelated changed paths Related paths are those which contain the path used to display the log. If you fetch the log for a folder, that means anything in that folder or below it. For a file it means just that one file. If you want to gray out the unrelated ones, check View → Gray unrelated changed paths Uncheck both menu items to hide the unrelated paths completely.

In the lower left there are the checkboxes All branches and Show whole project. Use these to override the branch resp. a file/folder filter and show the log for the whole repository. Please note that these settings are remembered for a repository - even if you started the log dialog on a single file.

You can show whole project history, no choose directory or file by click Show Whole Project

View+Labels → Tags View+Labels → Local branches View+Labels → Remote branches You can disable showing some reference types in the log graph.

View → Gravatar You can enable/disable Gravatar for a specific repository.

Walk Behavior → First Parent just follow up first parent commit. This will help understand the overall history.

Walk Behavior → No merges Skips all merge points.

Walk Behavior → Follow renames This is available to a single file only, which tracks renames. Otherwise, the log list stops at the commit that the current filename introduced.

Walk Behavior → Compressed Graph The log graph is simplified to include merge points, commits with references, and possibly other commits.

Walk Behavior → Show labeled commits only The log graph is simplified to include commits with references only.

2.14.7. Navigation

You can use the drop-down control on the upper right to select a navigation type (e.g. Author Email, Parent 1, Selection history), then use the Up and Down green buttons to navigate through the commits which match the navigation type relative to the current selected one.

Alternatively to the Up and Down green buttons, hotkeys **ALT+UP** and **ALT+DOWN** are also available.

Regarding the navigation type "Selection History", TortoiseGit memorizes the history of selected commits, so that you can navigate through those commits you selected in the past easily. You can also navigate them by pressing **ALT+LEFT**, **ALT+RIGHT**, **Browse Back**, and **Browse Forward**. **Back** and **Forward** buttons on mouse are also available.

If you also press **SHIFT** you can navigate through the selection history without selecting the last commits (i.e., just scrolling to and highlighting them). This helps you to navigate through commits and then select the commit(s) you really want to select (e.g. you can compare the current selected commit with the one you selected before).

If you want to jump to a commit with a particular hash, you may do so by pressing **Ctrl+V** or **Shift+Insert** (into any log dialog element other than the search box) to paste the hash from the clipboard. If it has the form of a valid commit hash, the log dialog will attempt to jump to it.

2.14.8. Statistical Information

The Statistics button brings up a box showing some interesting information about the revisions shown in the Log dialog. This shows how many authors have been at work, how many commits they have made, progress by week, and much more. Now you can see at a glance who has been working hardest and who is slacking ;-)

2.14.8.1. Statistics Page

This page gives you all the numbers you can think of, in particular the period and number of revisions covered, and some min/max/average values.

2.14.8.2. Commits by Author Page

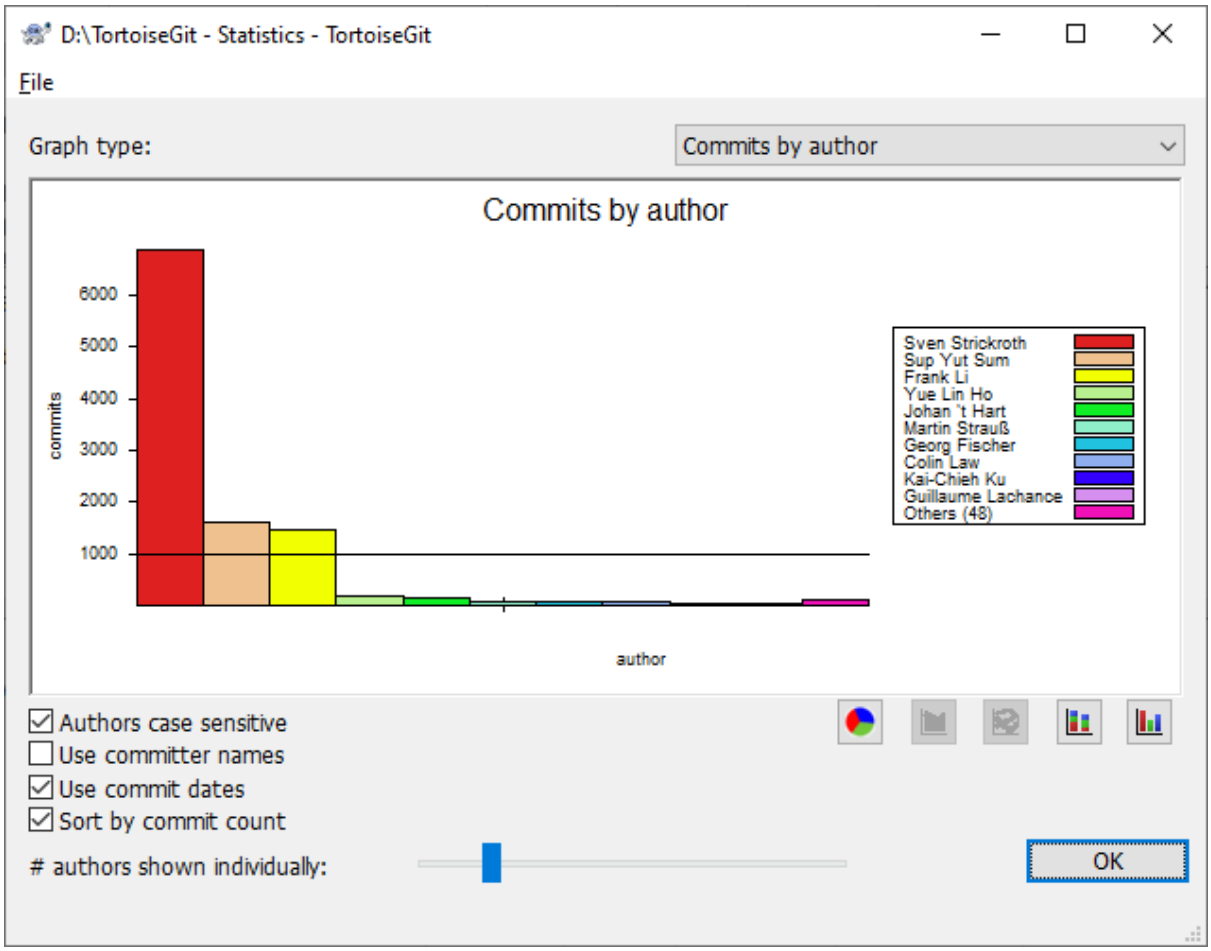


Figure 2.34. Commits-by-Author Histogram

This graph shows you which authors have been active on the project as a simple histogram, stacked histogram or pie chart.

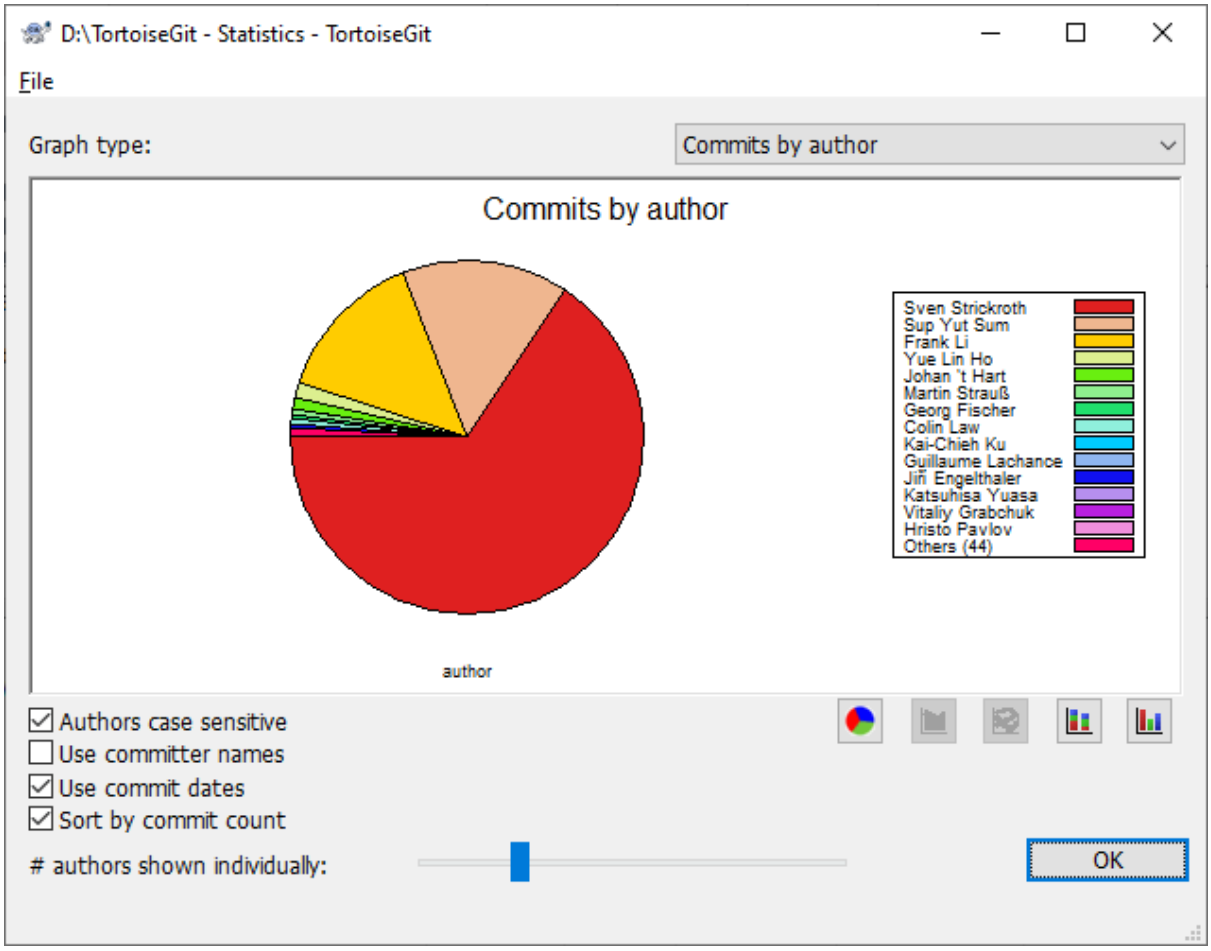


Figure 2.35. Commits-by-Author Pie Chart

Where there are a few major authors and many minor contributors, the number of tiny segments can make the graph more difficult to read. The slider at the bottom allows you to set a threshold (as a percentage of total commits) below which any activity is grouped into an *Others* category.

2.14.8.3. Commits by date Page

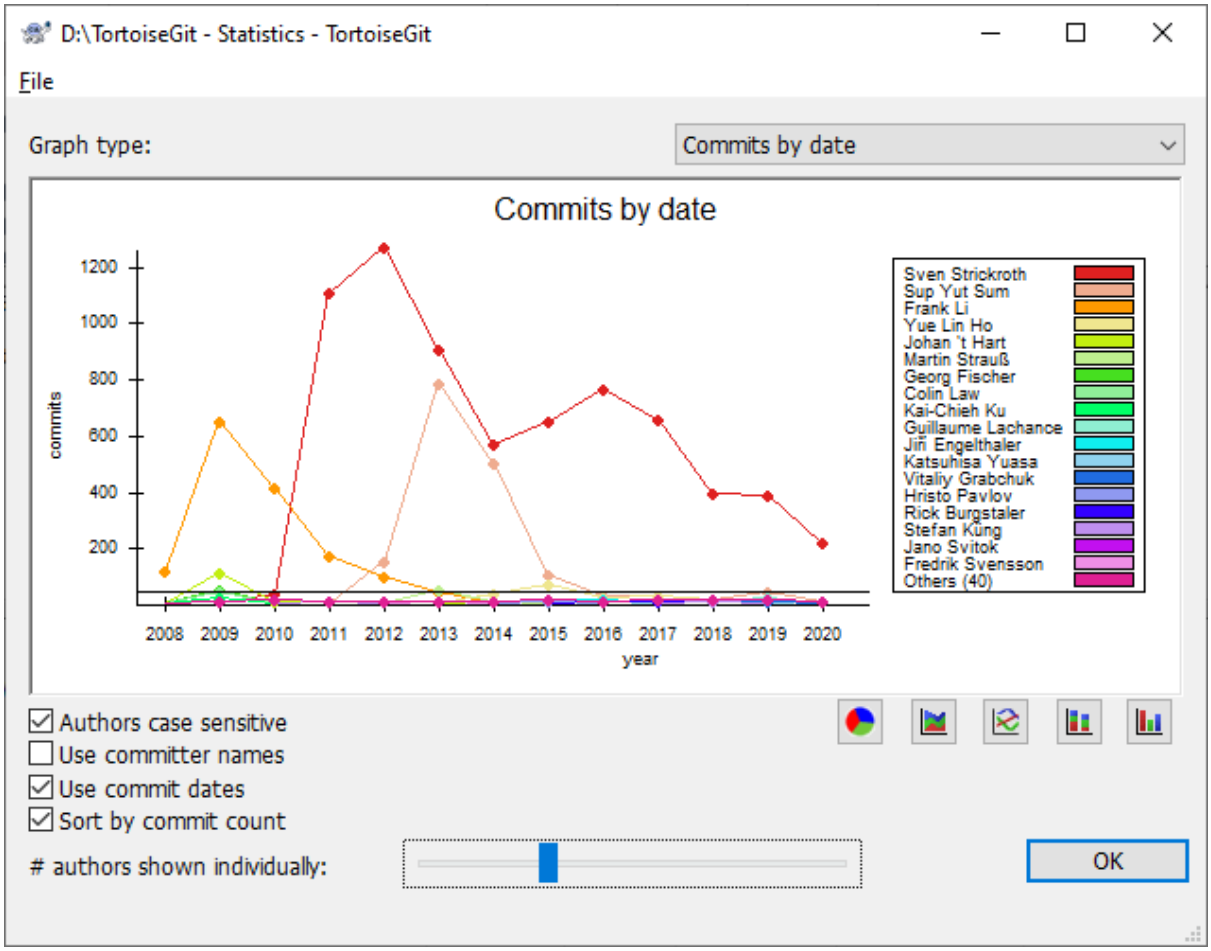


Figure 2.36. Commits-by-date Graph

This page gives you a graphical representation of project activity in terms of number of commits *and* author. This gives some idea of when a project is being worked on, and who was working at which time.

When there are several authors, you will get many lines on the graph. There are two views available here: *normal*, where each author's activity is relative to the base line, and *stacked*, where each author's activity is relative to the line underneath. The latter option avoids the lines crossing over, which can make the graph easier to read, but less easy to see one author's output.

By default the analysis is case-sensitive, so users `PeterEgan` and `PeteRegan` are treated as different authors. However, in many cases user names are not case-sensitive, and are sometimes entered inconsistently, so you may want `DavidMorgan` and `davidmorgan` to be treated as the same person. Use the Authors case insensitive checkbox to control how this is handled.

The statistics dialog also honors the `.mailmap` file (see Section G.4.9, “gitmailmap(5)”).

Note that the statistics cover the same period as the Log dialog. If that is only displaying one revision then the statistics will not tell you very much.

2.14.9. Refreshing the View

If you want to check the repository again for newer log messages, you can simply refresh the view using **F5**.

2.15. Revision Graphs

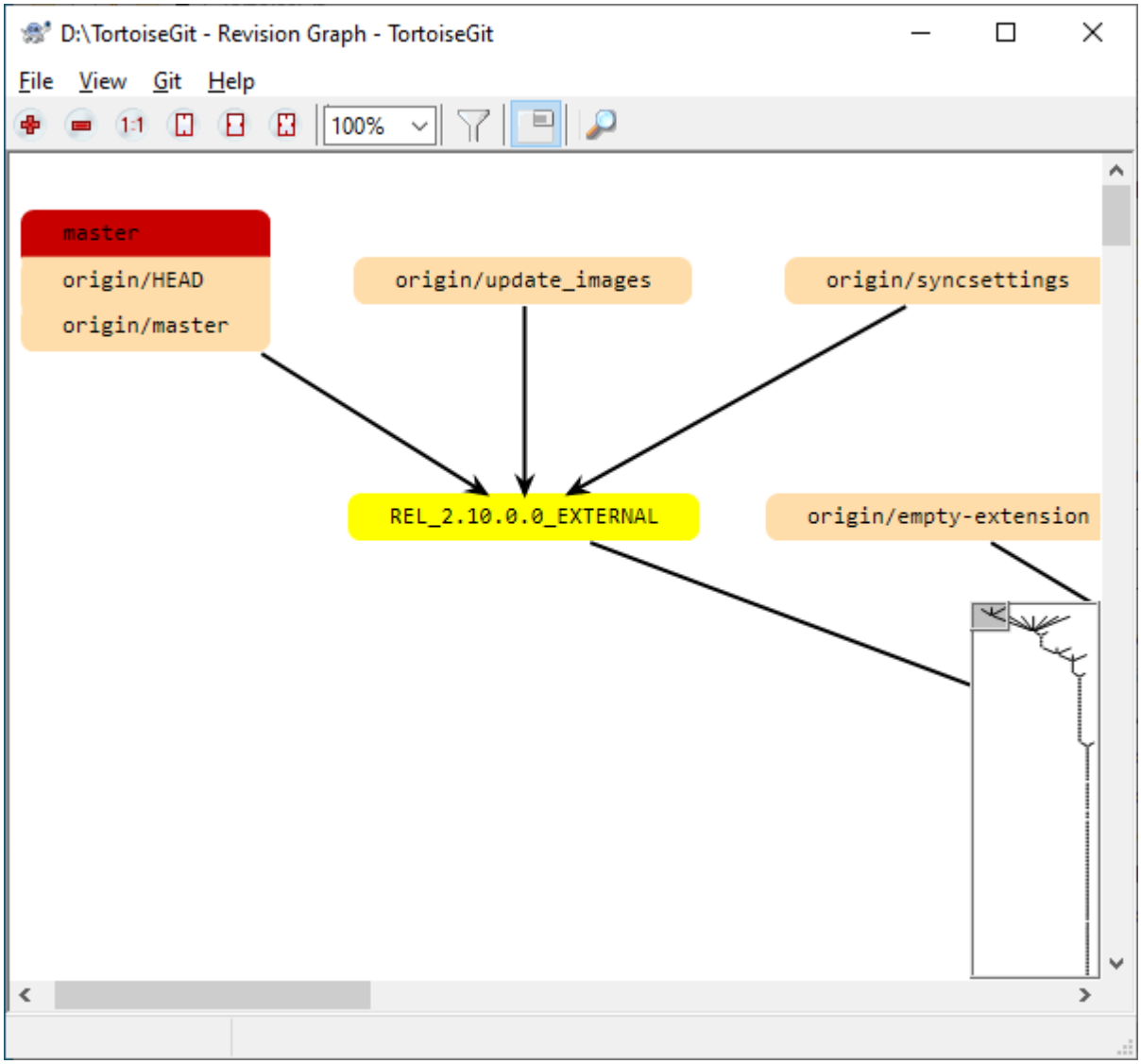


Figure 2.37. A Revision Graph

Sometimes you need to know where branches and tags were taken from the point, and the ideal way to view this sort of information is as a graph or tree structure. That's when you need to use TortoiseGit → Revision Graph...

This command analyses the revision history and attempts to create a direct graph showing the points at tag, branch and other reference.

> **Important**
>
> In order to generate the graph, TortoiseGit must fetch all log messages from the repository root. Just show commits which have some reference point to.

2.15.1. Revision Graph Nodes

Each revision graph node represents a revision in the repository where something changed in the tree you are looking at. Different types of nodes can be distinguished by color which can be configured using TortoiseGit → Settings

Note that the graph only shows the points at which items were reference by tag, branch or the other ref. Showing every revision of a project will generate a very large graph for non-trivial cases.

2.15.2. Using the Graph

To make it easier to navigate a large graph, use the overview window. This shows the entire graph in a small window, with the currently displayed portion highlighted. You can drag the highlighted area to change the displayed region.

The revision date, author and comments are shown in a hint box whenever the mouse hovers over a revision box.

If you select two revisions (Use **Ctrl**-left click), you can use the context menu to show the differences between these revisions. You can choose to show differences as at the branch creation points, but usually you will want to show the differences at the branch end points, i.e. at the HEAD revision.

You can view the differences as a Unified-Diff file, which shows all differences in a single file with minimal context. If you opt to Context Menu → Compare Revisions you will be presented with a list of changed files. Double click on a file name to fetch both revisions of the file and compare them using the visual difference tool.

If you right click on a revision you can use Context Menu → Show Log to view the history.

2.15.3. Refreshing the View

If you want to check the server again for newer information, you can simply refresh the view using **F5**.

2.16. Reference Log

The reference log (RefLog) displays the history of a reference (i.e., it is displayed to which commits it pointed in the past). It can be opened using TortoiseGit → RefLog, however, you have to hold the **Shift** key while right clicking on on a folder in the explorer in order to see this, because it is in the extended context menu by default.

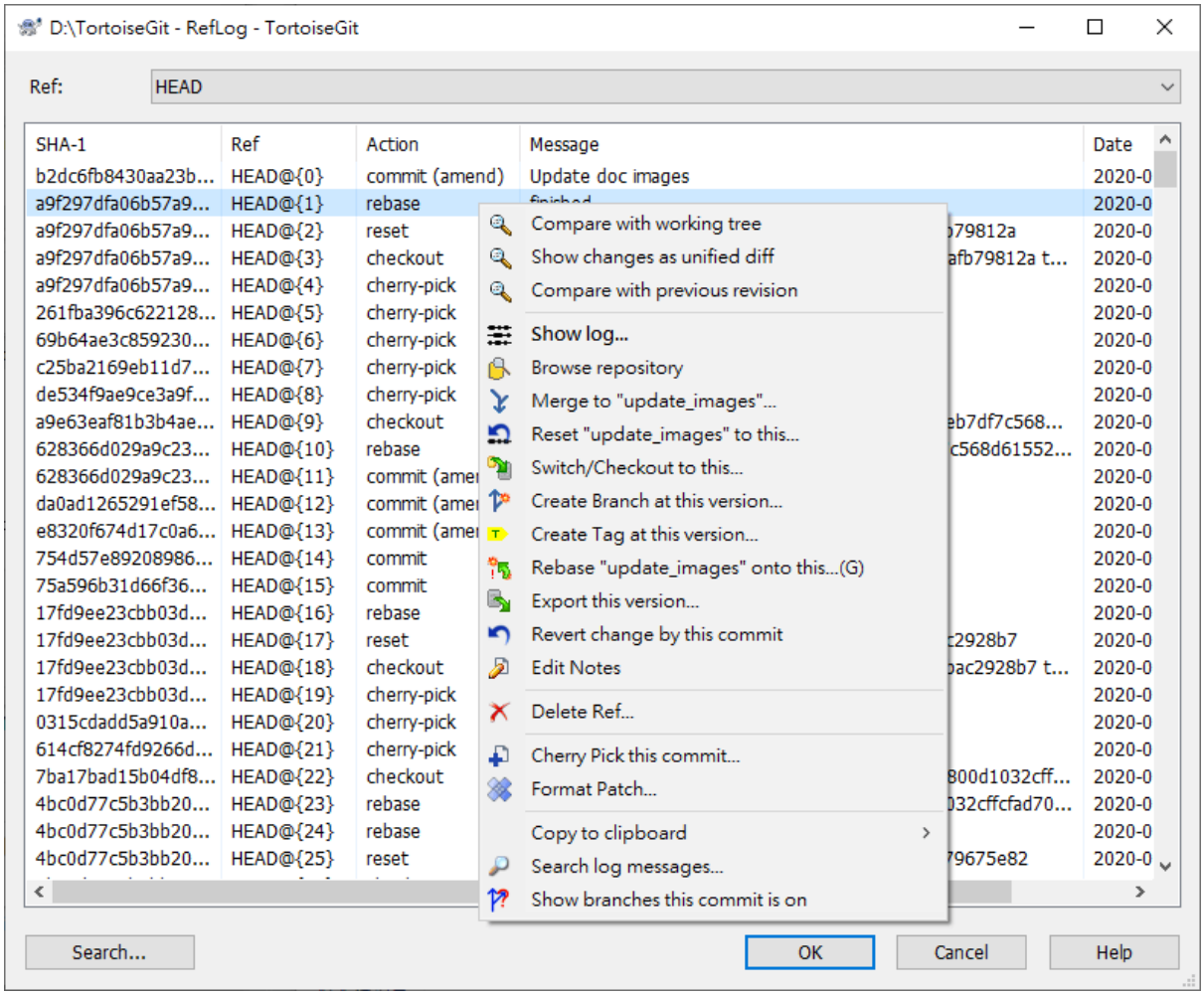


Figure 2.38. RefLog Dialog

The RefLog can be used to restore deleted commits or HEAD positions (e.g. when you deleted a branch which was HEAD some time ago).

You can find more information at Section G.3.109, “git-reflog(1)”.

2.17. The Repository Browser

Sometimes you need to see all contents/files of a repository, without having a working tree (e.g. a bare repository) or you want to see all files of a revision without switching to it. That's what the *Repository Browser* is for. You can open it using TortoiseGit → Repo-browser or from the log dialog (cf. Section 2.14, “Log Dialog”) using the context menu of a commit.

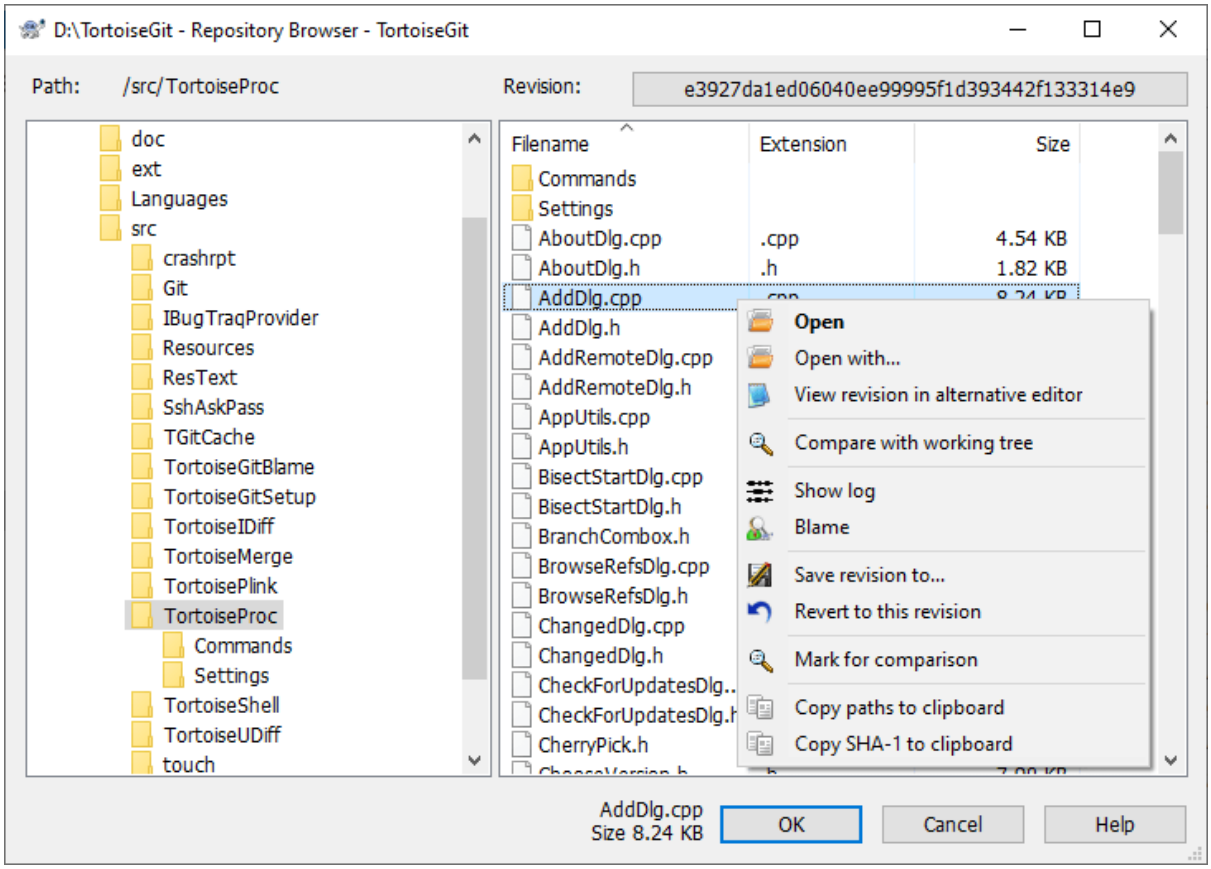


Figure 2.39. The Repository Browser

The repository browser looks very similar to the Windows explorer, except that it is showing the content of the repository at a particular revision rather than files on your computer. In the left pane you can see a directory tree, and in the right pane are the contents of the selected directory. At the top of the Repository Browser Window you can see the path within the repository and the revision you want to browse.

Just like Windows explorer, you can click on the column headings in the right pane if you want to set the sort order. And as in explorer there are context menus available in both panes.

In order to get an older version of a file you can click on a file and select Save revision to, but it is also possible to just drag one or more files into a Windows explorer window.

The context menu for a file allows you to:

- Open the selected file, either with the default viewer for that file type, or with a program you choose.
- Show the revision log for that file so you can see the history of it.
- Compare the file at the selected revision with the same file in your working tree.
- Blame the file, to see who changed which line and when.
- Save an unversioned copy of the file to your hard drive or revert this file in your working copy (i.e. saves the file to it's old path in the working tree).
- Copy the filename with full path shown in the address bar to the clipboard.

The context menu for a folder allows you to:

- Show the revision log for that folder.

- Copy the full path to the clipboard.

You can use **F5** to refresh the view as usual. This will refresh everything which is currently displayed.

2.18. Viewing Differences

One of the commonest requirements in project development is to see what has changed. You might want to look at the differences between two revisions of the same file, or the differences between two separate files. TortoiseGit provides a built-in tool named TortoiseGitMerge for viewing differences of text files. For viewing differences of image files, TortoiseGit also has a tool named TortoiseGitIDiff. Of course, you can use your own favorite diff program if you like.

2.18.1. File Differences

Local changes

If you want to see what (uncommitted) changes *you* have made in your working tree, just use the explorer context menu and select TortoiseGit → Diff.

Difference from a previous revision

If you want to see the difference between a particular revision and your working tree, use the Log dialog, select the revision of interest, then select Compare with working tree from the context menu (cf. Section 2.14, “Log Dialog”).

If you want to see the difference between the last committed revision and your working tree, assuming that the working tree hasn't been modified, just right click on the file. Then select TortoiseGit → Diff with previous version. This will perform a diff between the revision before the last-commit-date (as recorded in your working tree) and the working BASE. This shows you the last change made to that file to bring it to the state you now see in your working tree. It will not show changes newer than your working tree.

Difference between two previous revisions

If you want to see the difference between two revisions which are already committed, use the Log dialog and select the two revisions you want to compare (using the usual **Ctrl**-modifier). Then select Compare revisions from the context menu (cf. Section 2.14, “Log Dialog”). Then the Compare Revisions dialog appears, showing a list of changed files (maybe with a folder filter pre-applied). Read more in Section 2.18.3, “Comparing Version”.

All changes made in a commit

If you want to see the changes made to all files in a particular revision in one view, you can use Unified-Diff output (GNU patch format). This shows only the differences with a few lines of context. It is harder to read than a visual file compare, but will show all the changes together. From the Revision Log dialog select the revision of interest, then select Show Differences as Unified-Diff from the context menu.

Difference between files

If you want to see the differences between two different files, you can do that directly in explorer by selecting both files (using the usual **Ctrl**-modifier). Then from the explorer context menu select TortoiseGit → Diff.

Difference to another branch/tag

If you want to see the changes of different branches (maybe the current one to another branch or two branches) you can use the log dialog and select the two revisions as described above for "Difference between two previous revisions". An easier way is to open the reference browser (cf. Section 2.11, “Browse All Refs”). There you can click on one branch and select Compare to working tree to see all changes between that branch and your current state of the working tree. You can also select two branches and compare those using the context menu as described in Section 2.11, “Browse All Refs”.

Difference between folders

The built-in tools supplied with TortoiseGit do not support viewing differences between directory hierarchies.

If you have configured a third party diff tool, you can use **Shift** when selecting the Diff command to use the alternate tool resp. the build in tool. Read Section 2.36.4, "External Program Settings" to find out about configuring other diff tools.

2.18.2. Line-end and Whitespace Options

Sometimes in the life of a project you might change the line endings from `CRLF` to `LF`, or you may change the indentation of a section. Unfortunately this will mark a large number of lines as changed, even though there is no change to the meaning of the code. The options here will help to manage these changes when it comes to comparing and applying differences. You will see these settings in the Comparing Version dialog (cf. Section 2.18.3, "Comparing Version"), as well as in the settings for TortoiseGitMerge.

Ignore line endings excludes changes which are due solely to difference in line-end style.

Compare whitespaces includes all changes in indentation and inline whitespace as added/removed lines.

Ignore whitespace changes excludes changes which are due solely to a change in the amount or type of whitespace, e.g. changing the indentation or changing tabs to spaces. Adding whitespace where there was none before, or removing a whitespace completely is still shown as a change.

Ignore all whitespaces excludes all whitespace-only changes.

Naturally, any line with changed content is always included in the diff.

2.18.3. Comparing Version

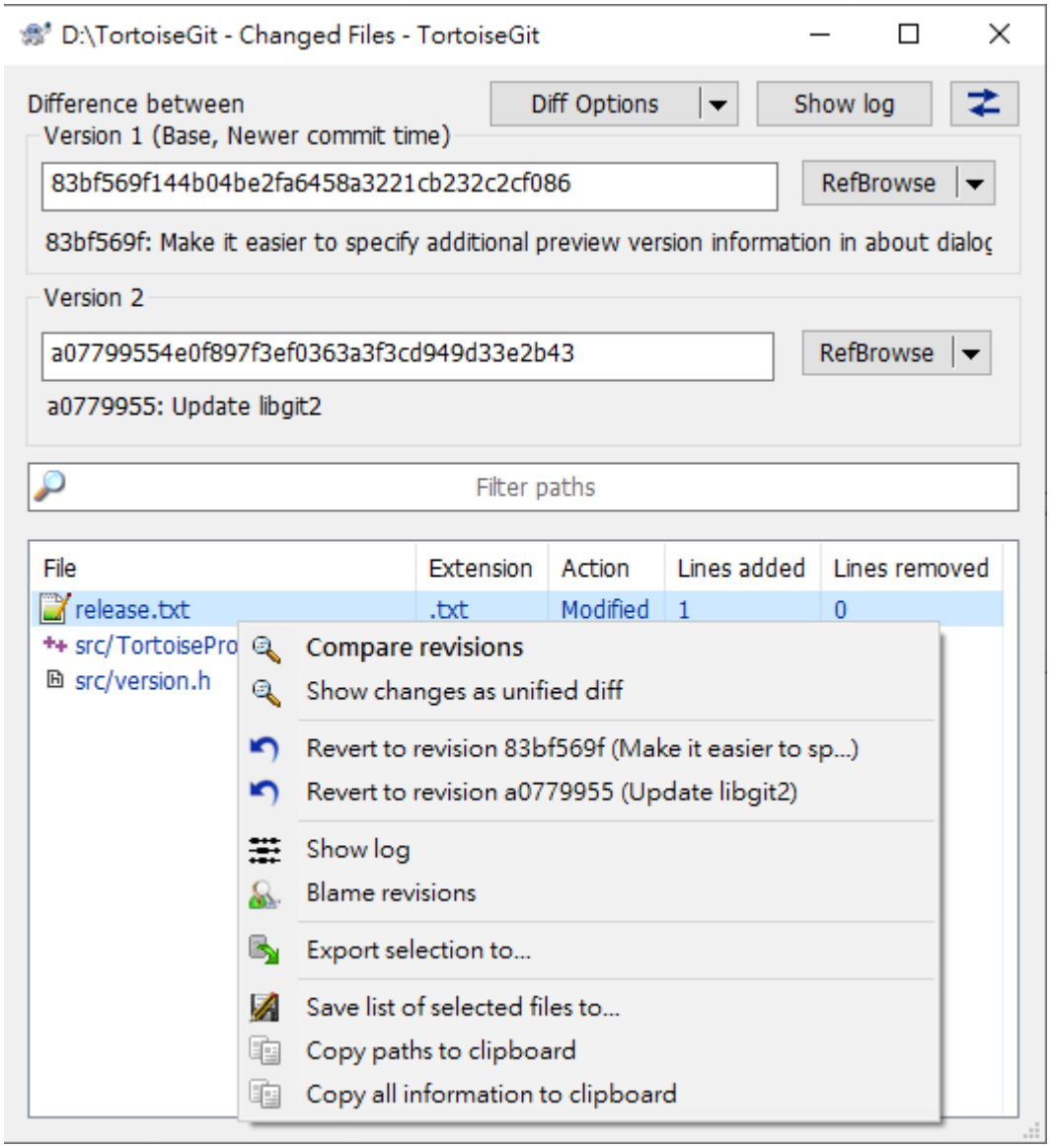


Figure 2.40. The Compare Revisions Dialog

In log dialog, when you select two commits Context menu → Compare revisions , or when you select a commit Context menu → Compare with previous version / Compare with working tree ; or in Windows Explorer, when you select no files or a folder TortoiseGit context menu → Diff with previous version , the Compare Revisions Dialog comes up.

This dialog shows a list of all files which have changed and allows you to compare them individually using context menu.

You can Revert selected files to version 1 or version 2. There are 2 menu items for this purpose. Context menu → Revert to revision xxxxxxx / Revert to revision yyyyyyy where xxxxxxx is revision 1 short hash and yyyyyyy revision is 2 short hash.

You can export a *change tree*, which is useful if you need to send someone else your project tree structure, but containing only the files which have changed. This operation works on the selected files only, so you need to select the files of interest - usually that means all of them - and then Context menu → Export selection to.... You will be prompted for a location to save the change tree.

You can also export the *list* of changed files to a text file using Context menu → Save list of selected files to....

If you want to export the list of files *and* the actions (modified, added, deleted) as well, you can do that using Context menu → Copy selection to clipboard.

The button at the top allows you to change the direction of comparison. You can show the changes need to get from A to B, or if you prefer, from B to A.

The buttons with the revision numbers on can be used to change to a different revision range. When you change the range, the list of items which differ between the two revisions will be updated automatically.

If the list of filenames is very long, you can use the search box to reduce the list to filenames containing specific text. Note that a simple text search is used, so if you want to restrict the list to C source files you should enter `.c` rather than `*.c`. The search syntax is similar to the one available in the Log dialog (cf. Section 2.14.6, "Filtering Log Messages").

2.18.4. Diffing submodules using Submodule Diff Dialog

The built-in diff command of git is available for diffing submodules, but we often find ourselves wanting to see more details how a submodule has changed too. That's why we created Submodule Diff Dialog. The Submodule Diff Dialog is only accessible using the Section 2.5, "Committing Your Changes To The Repository" or Section 2.6, "Getting Status Information" dialogs using the Compare with base entry in the context menu for a submodule.

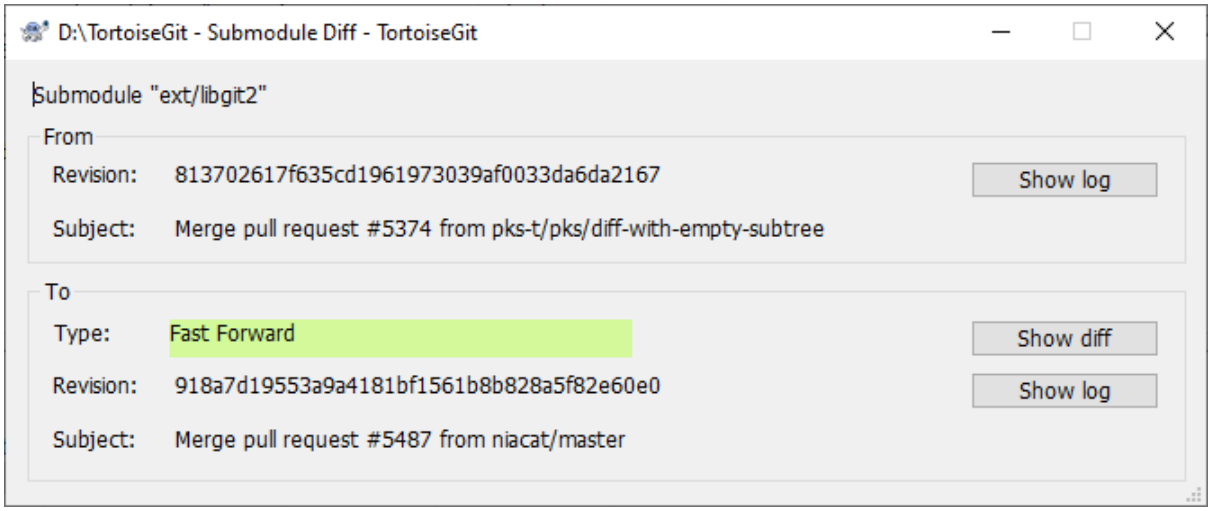


Figure 2.41. The submodule difference dialog

The 'From' group box on the top displays the information of the original revision. Below it, there is a 'To' group box, which display the information of the changed revision. For each group box, the full commit hash is displayed, and can be highlighted and copied to clipboard; the subject (i.e. first line of commit message) is displayed and also copyable to clipboard; the Show Log button brings you to a new Log Dialog and jump to that revision.

To better draw the attention of the change of revision of submodule mounted, we added some indicators. In 'To' group box, there is a change type field. Here list out the types:

Fast-forward

Topology-based. This is for a fast-forward change.

Rewind

Topology-based. This is the reversed direction of a fast-forward change.

Newer commit time

Time-based. If it is neither fast-forward nor rewind, then we compare commit time. This is for a revision with newer commit time than the original revision.

Older commit time

Time-based. This is the reversal of 'Newer commit time'.

Same commit time

Time-based. The commit time is the same. This may be produced by auto-generating commits or committing at the same time by two persons.

New Submodule

This is for newly added submodule.

Delete Submodule

This is for deleted submodule.

Unknown

This is for submodule revision hash not changed, error, etc..

If current workspace of the submodule is dirty, the commit hash will be rendered in yellow background and red text.

In both group boxes, if the revision is not fetched, submodule not initialized or other errors, the commit hash will be rendered in red background.

2.18.5. Diffing Images Using TortoiseGitIDiff

There are many tools available for diffing text files, including our own TortoiseGitMerge, but we often find ourselves wanting to see how an image file has changed too. That's why we created TortoiseGitIDiff.

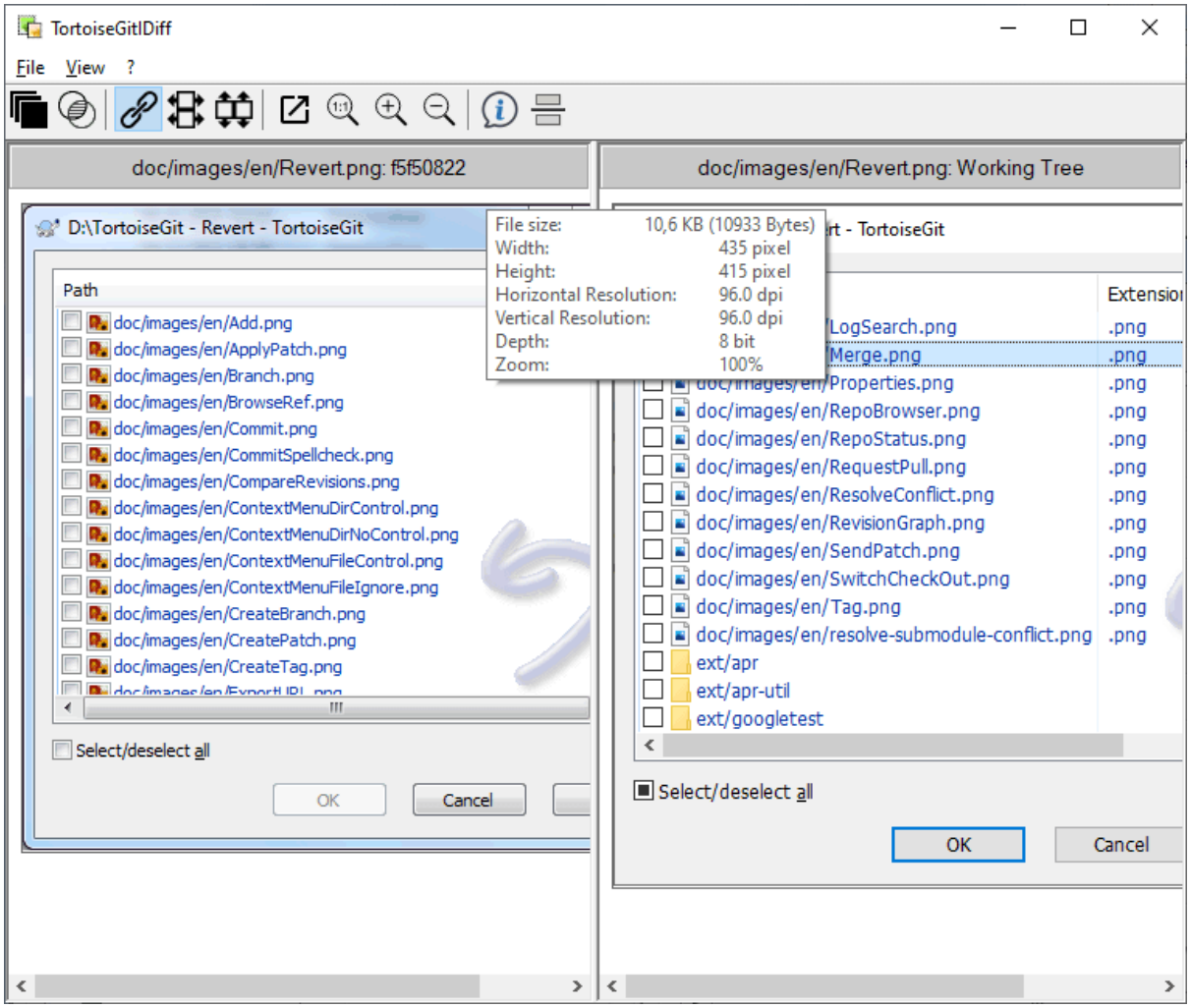


Figure 2.42. The image difference viewer

TortoiseGit → Diff for any of the common image file formats will start TortoiseGitIDiff to show image differences. By default the images are displayed side-by-side but you can use the View menu or toolbar to switch to a top-bottom view instead, or if you prefer, you can overlay the images and pretend you are using a lightbox.

Naturally you can also zoom in and out and pan around the image. You can also pan the image simply by left-dragging it. If you select the Link images together option, then the pan controls (scrollbars, mousewheel) on both images are linked.

An image info box shows details about the image file, such as the size in pixels, resolution and color depth. If this box gets in the way, use View → Image Info to hide it. You can get the same information in a tooltip if you hover the mouse over the image title bar.

When the images are overlaid, the relative intensity of the images (alpha blend) is controlled by a slider control at the left side. You can click anywhere in the slider to set the blend directly, or you can drag the slider to change the blend interactively. **Ctrl+Shift**-Wheel to change the blend.

The button above the slider toggles between two preset blends, indicated by the markers on either side of the blend slider. By default one is at the top and the other at the bottom, so the toggle button just switches between one image and the other. You can move the markers to choose the two blend values that the toggle button will use.

Sometimes you want to see a difference rather than a blend. You might have the image files for two revisions of a printed circuit board and want to see which tracks have changed. If you disable alpha blend mode, the difference will be shown as an *XOR* of the pixel color values. Unchanged areas will be plain white and changes will be colored.

2.18.6. External Diff/Merge Tools

If the tools we provide don't do what you need, try one of the many open-source or commercial programs available. Everyone has their own favorites, and this list is by no means complete, but here are a few that you might consider:

WinMerge

WinMerge [https://winmerge.org/] is a great open-source diff tool which can also handle directories.

Perforce Merge

Perforce is a commercial RCS, but you can download the diff/merge tool for free. Get more information from *Perforce* [https://www.perforce.com/products/helix-core-apps/merge-diff-tool-p4merge].

KDiff3

KDiff3 is a free diff tool which can also handle directories. You can download it from *here* [http://kdiff3.sf.net/].

ExamDiff

ExamDiff Standard is freeware. It can handle files but not directories. ExamDiff Pro is shareware and adds a number of goodies including directory diff and editing capability. In both flavours, version 3.2 and above can handle unicode. You can download them from *PrestoSoft* [https://www.prestosoft.com/].

Beyond Compare

Similar to ExamDiff Pro, this is an excellent shareware diff tool which can handle directory diffs and unicode. Download it from *Scooter Software* [http://www.scootersoftware.com/].

Araxis Merge

Araxis Merge is a useful commercial tool for diff and merging both files and folders. It does three-way comparison in merges and has synchronization links to use if you've changed the order of functions. Download it from *Araxis* [https://www.araxis.com/merge/index.html].

SciTE

This text editor includes syntax coloring for unified diffs, making them much easier to read. Download it from *Scintilla* [https://www.scintilla.org/SciTEDownload.html].

Notepad2

Notepad2 is designed as a replacement for the standard Windows Notepad program, and is based on the Scintilla open-source edit control. As well as being good for viewing unified diffs, it is much better than the Windows notepad for most jobs. Download it for free *here* [https://www.flos-freeware.ch/notepad2.html].

Notepad2 is included in the TortoiseGit Setup as an alternative editor which support LF only line endings. An entry in the start menu named Notepad2 is created.

Read Section 2.36.4, “External Program Settings” for information on how to set up TortoiseGit to use these tools.

2.19. Adding New Files

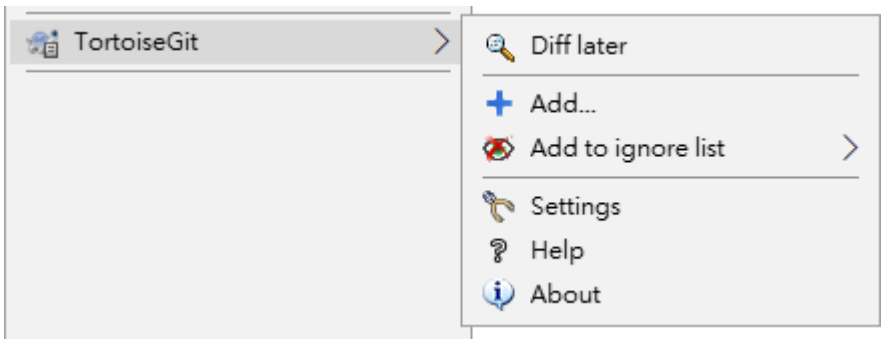


Figure 2.43. Explorer context menu for unversioned files

If you created new files during your development process then you need to add them to source control too. Select the file(s) and/or NOT empty directory and use TortoiseGit → Add.

After you added the files to source control the file appears with a `added` icon overlay which means you first have to commit (and push) your working tree to make those files available to other developers. Just adding a file does *not* affect any remote repository!

Many Adds

You can also use the Add command on folders. In that case, the add dialog will show you all unversioned files inside that versioned folder. This helps if you have many new files and need to add them all at once.

Empty directories

Git only tracks content and, thus, cannot version (empty) directories. If you need a directory to be automatically created on checkout, make sure at least one versioned file is in it (e.g. a placeholder file such as `.gitkeep` or `.gitignore`).

To add files from outside your working tree you can use the drag-and-drop handler:

1. select the files you want to add
2. right-drag them to the new location inside the working tree
3. release the right mouse button
4. select Context Menu → Git copy and add files to this WC. The files will then be copied to the working tree and added to version control.

You can also add files within a working tree simply by (left-)dragging and dropping them onto the commit dialog.

If you add a file by mistake, you can undo the addition before you commit using TortoiseGit → Delete (keep local)... or Revert.

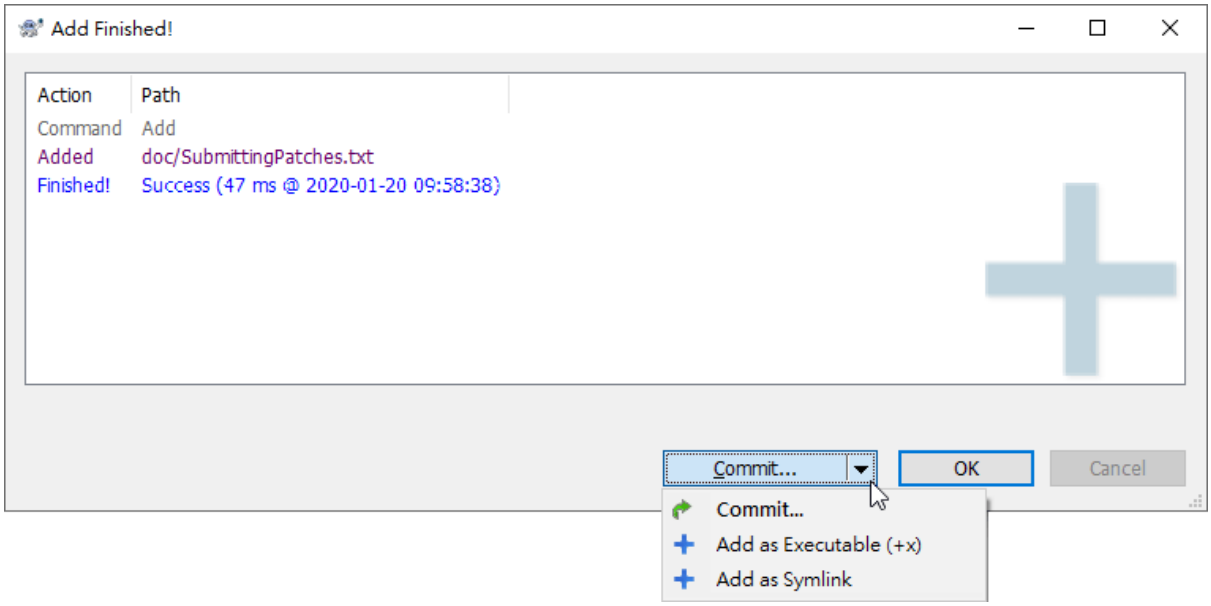


Figure 2.44. Add finished

After adding the files, you may commit by clicking the shortcut menu button. Additionally, there are options to mark the newly added files as executable / symlink. Remember to mark executable bit for files such as Unix shell script. This is to facilitate sharing repository with Linux / MacOS environment.

You can find more information at Section G.3.2, “git-add(1)”

2.20. Ignoring Files And Directories

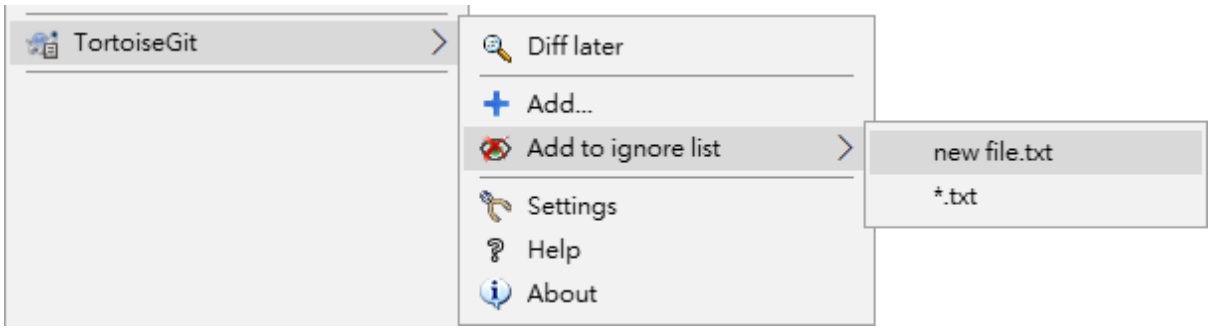


Figure 2.45. Explorer context menu for unversioned files

In most projects you will have files and folders that should not be subject to version control. These might include files created by the compiler, `*.obj`, `*.lst`, maybe an output folder used to store the executable, `bin/`, `obj/`. More examples include user-specific workspace settings `*.suo`, `*.user` (Visual Studio), backup files `*.bak`, `Backup*/`, Shell metadata files `Thumbs.db`, `Desktop.ini`, `.DS_Store/`. Whenever you commit changes, TortoiseGit shows your unversioned files, which fills up the file list in the commit dialog. Of course you can turn off this display, but then you might forget to add a new source file.

The best way to avoid these problems is to add the derived files to the project's ignore list. That way they will never show up in the commit dialog, but genuine unversioned source files will still be flagged up.

If you right click on one or more unversioned files, and select the command TortoiseGit → Add to Ignore List from the context menu, a submenu appears allowing you to select ignore by names or by extensions. Ignore dialog shows that allows you to select ignore type and ignore file.

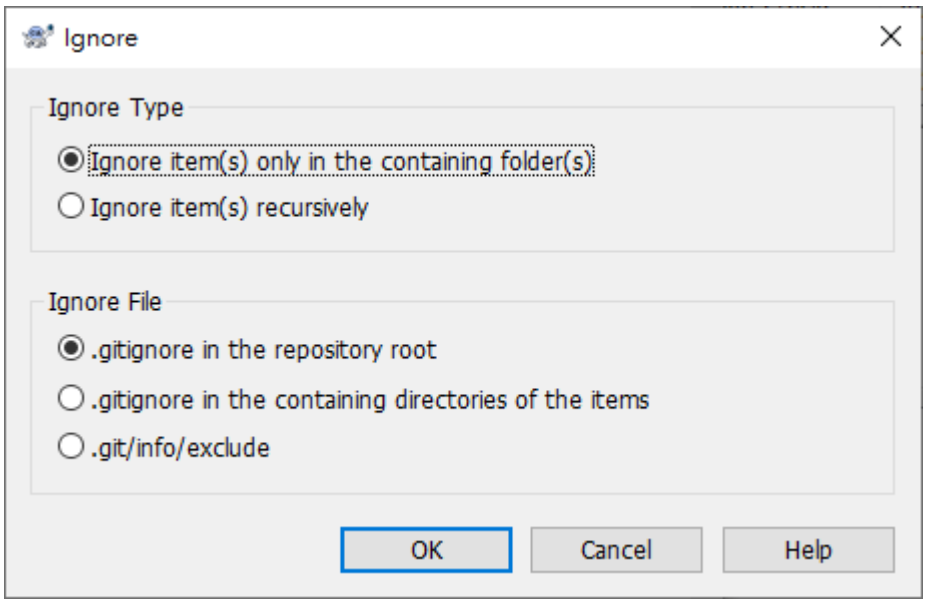


Figure 2.46. Ignore dialog

Ignore Type

Ignore item(s) only in containing folder(s)

Only ignore the selected pattern(s) within that folder(s).

Ignore item(s) recursively

Ignore items with the selected pattern(s) in that folder(s) and child folder(s).

Ignore File

.gitignore in the repository root

Write the ignore entries in .gitignore in the repository root. This allows you to synchronize the ignore list with remote repository.

.gitignore in the containing directories of the items

Write the ignore entries in .gitignore in the containing directories of the items. This allows you to synchronize the ignore list with remote repository.

.git/info/exclude

Write the ignore entries in .git/info/exclude in repository metadata. This allows you to store the ignore list locally, but cannot synchronize with remote repository.

If you want to remove one or more items from the ignore list, in current version of TortoiseGit, you have to manually edit the ignore list file using a text editor that can handle Unix EOL. That allows you to specify more general patterns using filename globbing, described in the section below. Read Section G.4.5, “gitignore(5)” for more information. Please be aware that each ignore pattern has to be placed on a separate line. Separating them by spaces does not work.

2.20.1. Pattern Matching in Ignore Lists

Git's ignore patterns make use of filename globbing, a technique originally used in Unix to specify files using meta-characters as wildcards. The following characters have special meaning:

*

Matches any string of characters, including the empty string (no characters).

?

Matches any single character.

[...]

Matches any one of the characters enclosed in the square brackets. Within the brackets, a pair of characters separated by "-" matches any character lexically between the two. For example `[AGm-p]` matches any one of `A`, `G`, `m`, `n`, `o` or `p`.

Pattern matching is case sensitive, which can cause problems on Windows. You can force case insensitivity the hard way by pairing characters, e.g. to ignore `*.tmp` regardless of case, you could use a pattern like `*.[Tt][Mm][Pp]`.

2.21. Deleting, Copying, Moving and Renaming Files and Folders

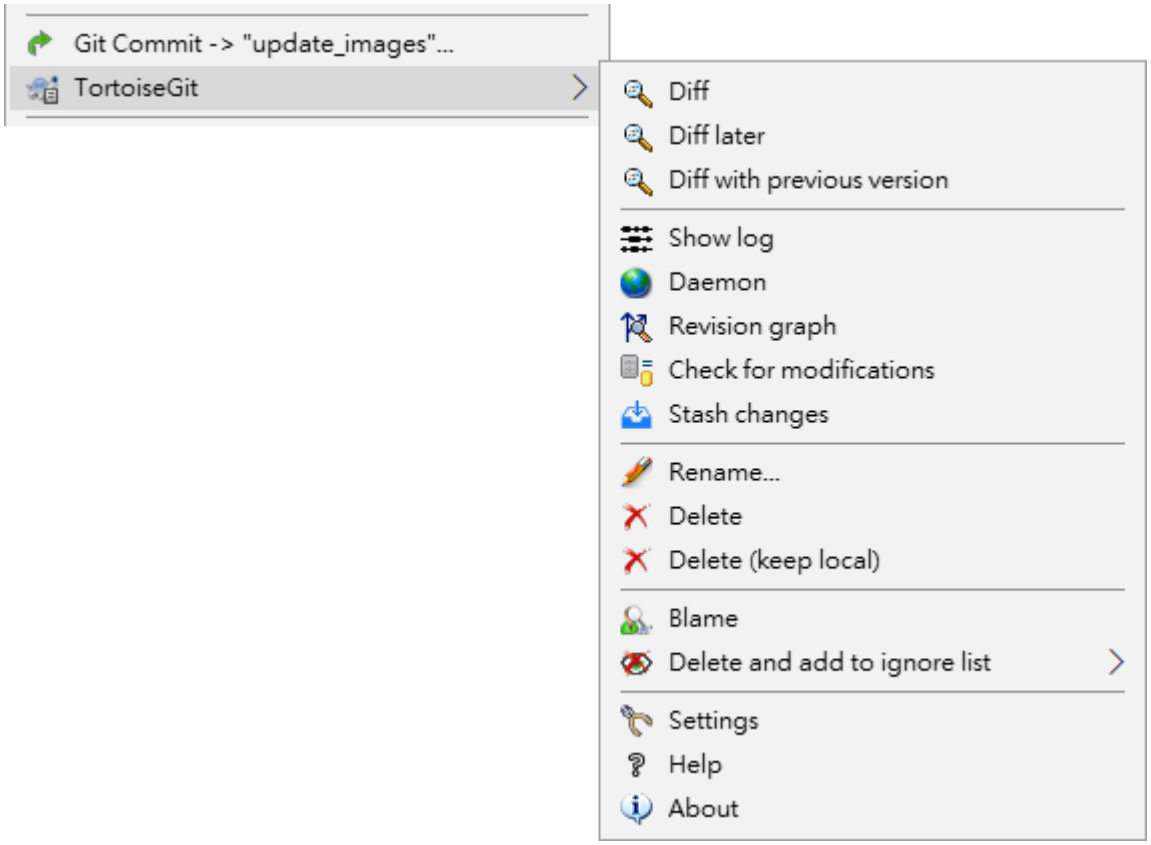


Figure 2.47. Explorer context menu for versioned files

As you work on a project, you will eventually need to manage files or folders, i.e. delete, copy, rename, or move them. TortoiseGit provides support for such operations. They are available from the context menu of versioned files (and folders) and are described in more detail in the following subsections.

Git only tracks content

Unlike Subversion and other version control systems, Git does *not* track file names. Git only tracks the contents of files and, therefore, it is not necessary to explicitly record copies, renames or moves. So there is generally no need to use Git's rename or remove functionality or even to "repair renames" as in Subversion. Renames and copies are automatically detected when you view the log. Nevertheless, adding (cf. Section 2.19, "Adding New Files") and removing files from version control need to be recorded.

Cannot copy between repositories

Whilst you can copy and files and folders *within* a repository, you *cannot* copy or move from one repository to another while preserving history using Git or TortoiseGit. Not even if the repositories live on the same server. All you can do is copy the content in its current state and add it as new content to the second repository.

2.21.1. Deleting files and folders

Use TortoiseGit → Delete to remove files or folders from Git.

When you use TortoiseGit → Delete on a file, it is removed from your working tree immediately as well as being marked for deletion in the repository on the next commit (i.e., the file is removed from the Git index). When you perform this action on a folder, all files/folders will be removed recursively. Up until you commit the change, you can get the file back using TortoiseGit → Revert on the parent folder or use the commit (cf. Section 2.5, "Committing Your Changes To The Repository") or status (cf. Section 2.6, "Getting Status Information") dialogs.

If you want to delete an item from version control, but keep it locally as an unversioned file/folder, use Extended Context Menu → Delete (keep local). There also is a special command for deleting an item and adding it to the ignore list (Extended Context Menu → Delete and add to ignore list, cf. Section 2.20, "Ignoring Files And Directories"). You may have to hold the **Shift** key while right clicking on the item in the explorer list pane (right pane) in order to see these options in the extended context menu.

If a file is deleted via the explorer instead of using TortoiseGit's context menu, the commit dialog (cf. Section 2.5, "Committing Your Changes To The Repository") and status (cf. Section 2.6, "Getting Status Information") dialogs will show the file as *missing*. Missing files are also automatically marked as deleted on commit when they are selected, but explicitly deleting a version-controlled file using TortoiseGit → Delete may make your intention on commit more clear while your are preparing the commit.

Getting a deleted file or folder back

If you have deleted a file or a folder and already committed that delete operation to the repository, then a normal TortoiseGit → Revert can't bring it back anymore. But the file or folder is not lost at all. If you know the revision the file or folder got deleted (if you don't, use the log dialog to find out) open the repository browser and switch to that revision. Then select the file or folder you deleted, right-click and select Context Menu → Revert to this revision. Refer to Section 2.17, "The Repository Browser" and Section 2.14, "Log Dialog" to find out more.

2.21.2. Copying Files and Folders

It often happens that you already have the files you need in another folder in your repository. As Git only tracks content, simply copy the items across your working tree and mark them as added or select them while committing (cf. Section 2.19, "Adding New Files").

In order to get older versions of a file you can use the repository browser or the log dialog to locate content you want, and copy (or drag and drop) it into your working tree directly from the repository. Refer to Section 2.17, "The Repository Browser" and Section 2.14, "Log Dialog" to find out more.

2.21.3. Moving Files and Folders

If you want to do a simple in-place rename of a file or folder, use Context Menu → Rename... Enter the new name for the item and you're done. You can move an item around in the whole working tree, i.e., the name may start with `..\` to move a file upwards in the folder structure. Use the ... to open a file chooser dialog which automatically generates the correct destination name.

The easiest way to move files and folders from within a working tree is to use the right-drag menu:

1. Select the files or directories you want to move

2. right-drag them to the new location inside the working tree
3. release the right mouse button
4. in the popup menu select Context Menu → Git Move versioned files here

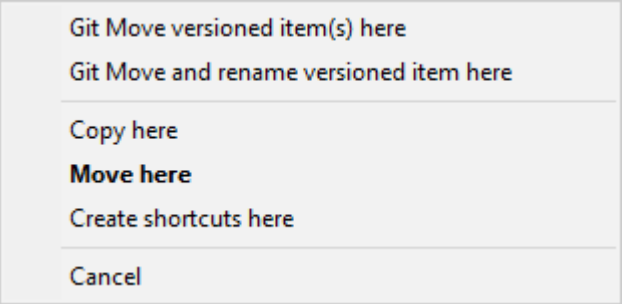


Figure 2.48. Right drag menu for a directory under version control

Since Git only tracks content and not filenames, you could also simply move/rename files in Explorer. But then you have to mark the old file as deleted, and mark the new file as added separately later (at the latest during commit).

Moving Submodules

Moving Submodules (Section G.3.141, "git-submodule(1)") using TortoiseGit's Move and Rename commands should work. There are, however, notes for the used Section G.3.123, "git-rm(1)" command (cf. the section called "SUBMODULES").

2.21.4. Changing case in a filename

Making case-only changes to a filename needs special attention, because Windows does not honor the filename casing by default. Therefore just renaming a file using the rename command of the Explorer is likely not to work. It is important to rename it using Git in order to update the index to make it use the new filename. Use the Rename... command in the TortoiseGit submenu.

2.21.5. Dealing with filename case conflicts

If the repository already contains two files with the same name but differing only in case (e.g. `TEST.TXT` and `test.txt`), you will not be able to commit, and only one of them can be checkout on a Windows client. Whilst Git (in general) supports case-sensitive filenames, Windows does not.

This sometimes happens when files are committed from a system with a case-sensitive file system, like Linux, or when the setting `core.ignorecase` is set to false (cf. Section G.3.29, "git-config(1)").

In that case, you have to decide which one of them you want to keep and delete the other(s) from the repository (or rename the other(s)). Easiest way is to do that on a case-sensitive file system, followed by committing and pushing the changes. Doing it on Windows requires several steps (and two commits):

Solution

1. Delete the file in explorer.

Caution

Do NOT use the Delete or the Delete (keep local) command in the TortoiseGit submenu!

2. Open the Commit dialog. (All the checked items are of `Deleted` status.)
3. Uncheck only one item you want to keep.

4. Commit the changes.

5. Revert deletion of the wanted file in order to get it back. If you want to keep both or more files which had the "same" name, but with a different new name, do this for all files in question and rename them before proceeding with the next file.

2.21.6. Deleting Unversioned Files

Usually you set your ignore list such that all generated files are ignored in Git. But what if you want to clear all those ignored items to produce a clean build? Usually you would set that in your makefile, but if you are debugging the makefile, or changing the build system it is useful to have a way of clearing the decks.

TortoiseGit provides just such an option using Extended Context Menu → Clean up.... You may have to hold the **Shift** while right clicking on a folder in the explorer list pane (right pane) in order to see this in the context menu. This will show a dialog which lists all possible clean up options (cf. Section 2.23, “Cleanup”).

2.22. Undo Changes

If you want to undo all changes you made in a file since your last commit you need to select the file, right click to pop up the context menu and then select the command TortoiseGit → Revert A dialog will pop up showing you the files that you've changed and can revert. Select those you want to revert and click on OK.

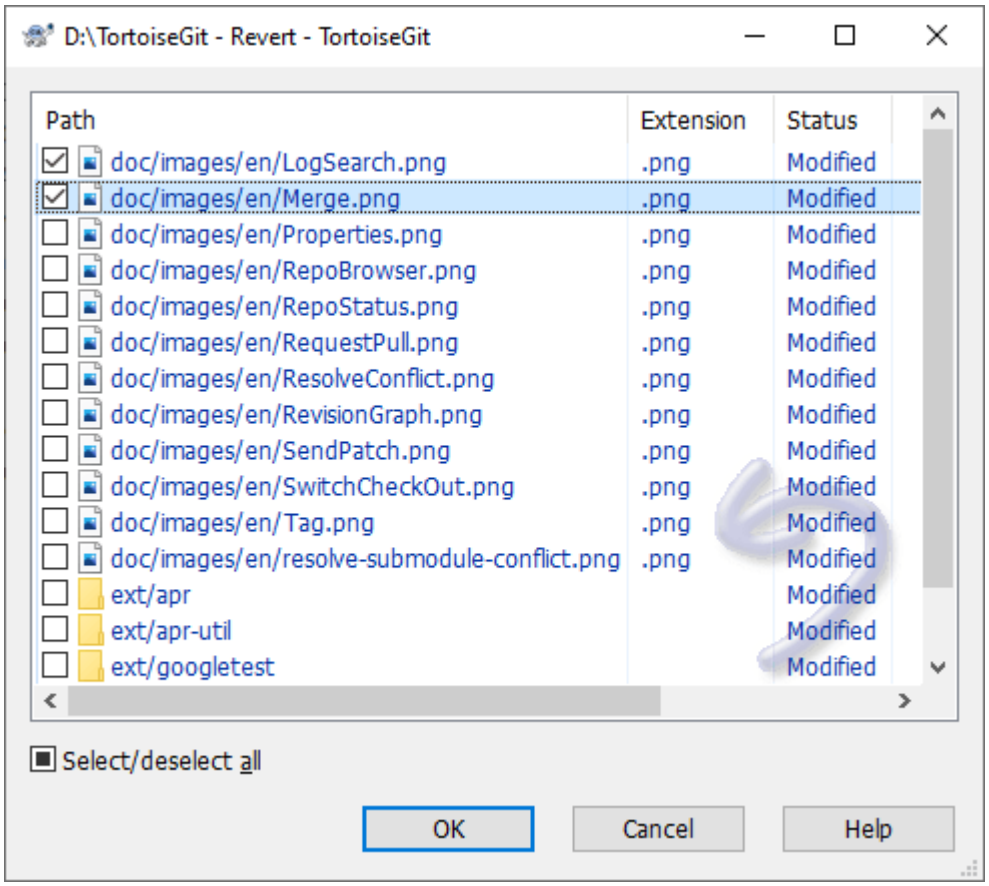


Figure 2.49. Revert dialog

If you want to undo a deletion or a rename, you need to use Revert on the parent folder (or commit or repository status dialog) as the deleted item does not exist for you to right-click on.

If you want to undo the addition of an item, this appears in the context menu as TortoiseGit → Delete (keep local). This is really a revert as well, but the name has been changed to make it more obvious.

The columns in this dialog can be customized in the same way as the columns in the Check for modifications dialog. Read Section 2.6.3, “Status” for further details.

Undoing Changes which have been committed

Revert will only undo your local changes. It does *not* undo any changes which have already been committed. If you want to undo all the changes which were committed in a particular revision, read Section 2.14, “Log Dialog” and Section 2.17, “The Repository Browser” for further information.

Reverting a whole commit

If you want to undo a whole commit, then you should use the log dialog and select Revert change by this commit on a revision/commit (cf. Section 2.14, “Log Dialog”). Then all changes of this commit are undone and a revert commit is created which need to be committed manually (cf. Section G.3.122, “git-revert(1)”). It is also possible to (hard) reset to a previous commit, then all commits after that are forgotten (cf. Section 2.24, “Reset”) - this might not be recommended if the changes are already pushed (also see https://stackoverflow.com/q/27032850/3906760).

Reverting to an old version of a file

If you want to revert a file to an older version open the Log Dialog (cf. Section 2.14, “Log Dialog”) on the file (or the folder if the file was deleted). Then select the specific commit you want to revert to. Now you have two options: 1) Select the specific file entry in the file list in the bottom and select Revert to this revision resp. Revert to parent revision in the context menu of the file entry. 2) Select Browse Repository (cf. Section 2.17, “The Repository Browser”) in the context menu of the revision. Then navigate to the file and either select Save revision to... or drag the file to the location where you want to use it.

Revert is Slow

When you revert changes you may find that the operation takes a lot longer than you expect. This is because the modified version of the file is sent to the recycle bin, so you can retrieve your changes if you reverted by mistake. However, if your recycle bin is full, Windows takes a long time to find a place to put the file. The solution is simple: either empty the recycle bin or deactivate the Use recycle bin when reverting box in TortoiseGit's settings.

Revert != "git revert" for files

In the TortoiseGit naming a "revert" on a file is comparable to `git checkout HEAD -- filename` (or `git checkout REVISION -- filename`) for resetting a file to it's last (or a specific) committed state. This has nothing to do with Section G.3.122, “git-revert(1)”!

Section G.3.122, “git-revert(1)” is only referred to by Revert change by this commit in log dialog (cf. Section 2.14, “Log Dialog”).

2.23. Cleanup

In order to remove untracked or ignored files from the working tree use TortoiseGit → Cleanup. Then a dialog comes up which allows you to clean up the working tree by recursively removing files that are not under version control or ignored, starting from the current directory *or* on the whole working tree (depends on version of installed git).

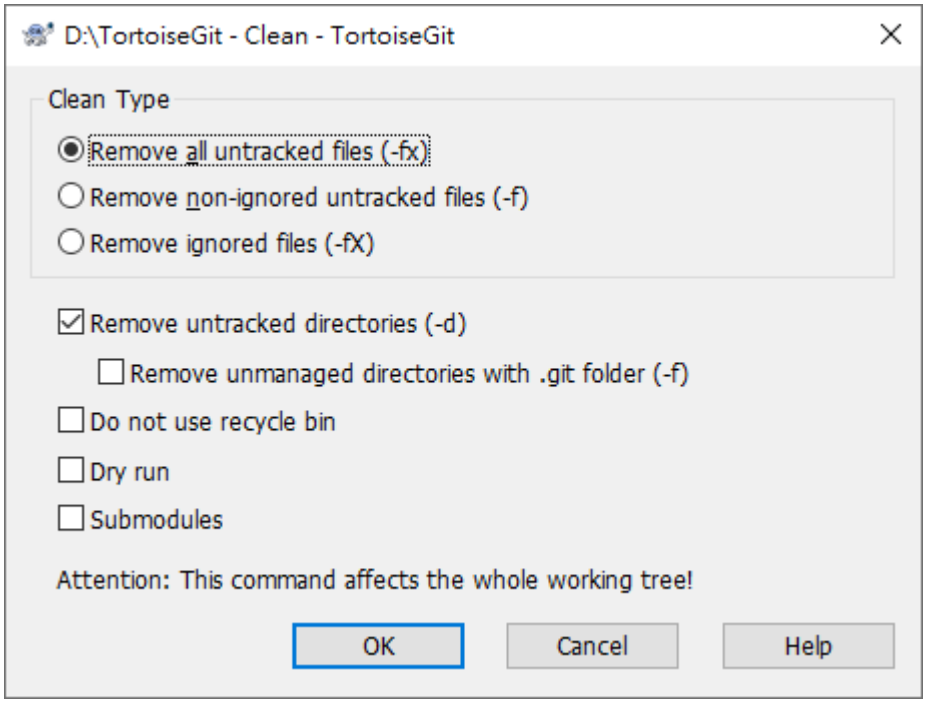


Figure 2.50. Clean dialog

Clean all untracked files This removes all untracked files, including those ignored by Git. This is the cleanest option.

Clean only non-ignore untracked files This removes untracked files, but excluding those ignored by Git.

Clean only ignored files This removes only files ignored by Git.

Remove untracked directories This removes untracked directories too.

Do not use recycle bin Use this option if you want to delete those files directly and permanently. Make sure you do not regret!

Dry run This just gives the list of files to be deleted, but it does not perform any deletion.

Submodules This also cleans submodules recursively.

You can find more information at Section G.3.23, “git-clean(1)”.

2.24. Reset

The reset dialog can be used to reset the current HEAD to the specified state and optionally also the index and the working tree. This can also be used to abort a merge.

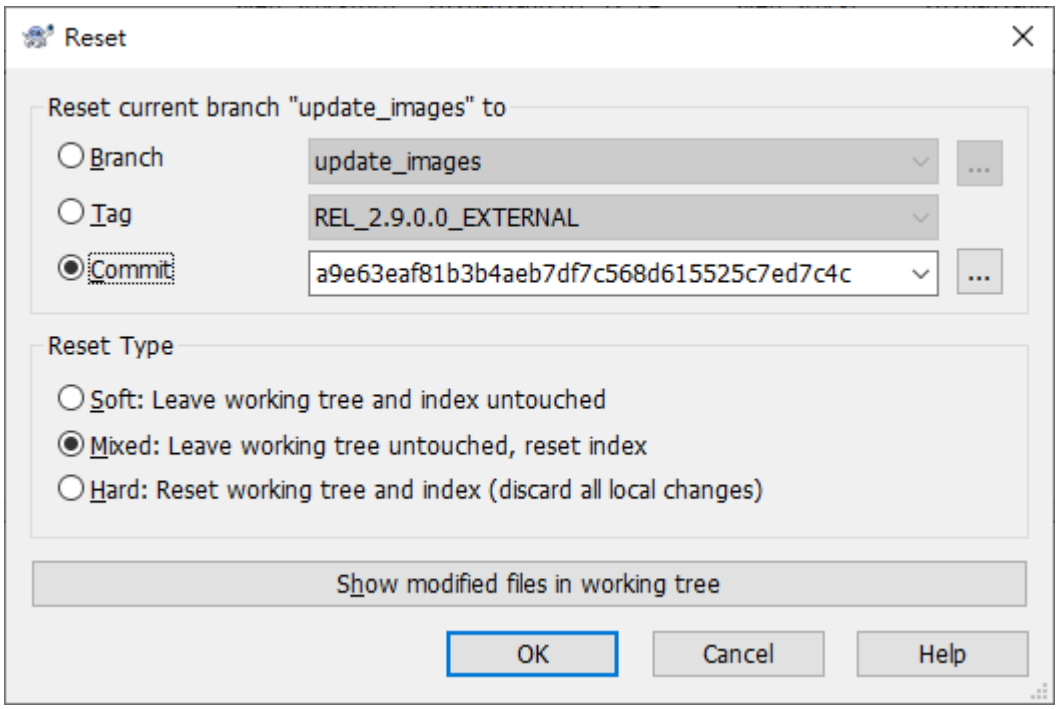


Figure 2.51. The Reset dialog

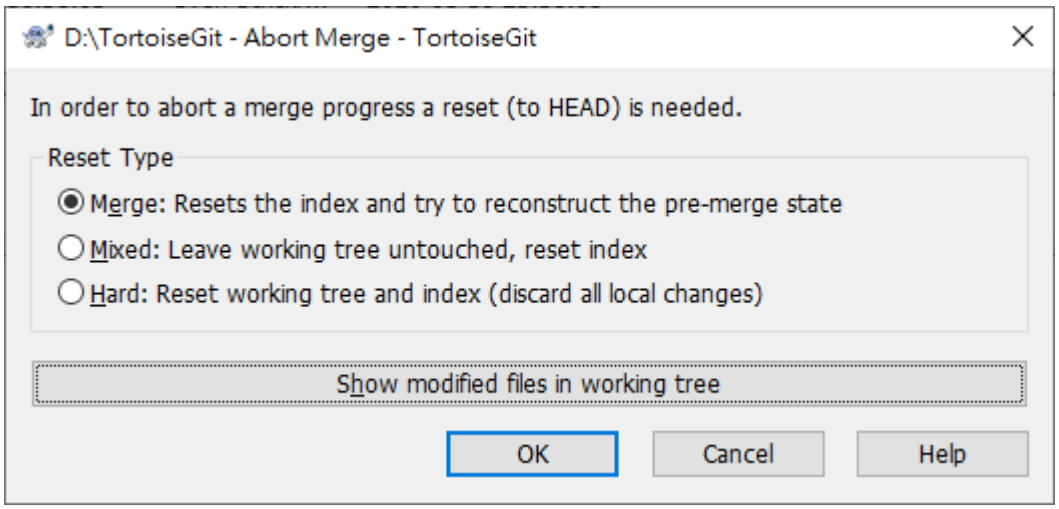


Figure 2.52. The Abort Merge dialog

On the Reset dialog, you can click ... to browse the log and choose a specific version. In Abort merge dialog, you can only reset to HEAD.

Soft: Leave working tree and index untouched Does not touch the index file nor the working tree at all (but resets the head to the selected commit, just like all modes do). This leaves all your changed files "Changes to be committed" as before. This option is not available in Abort Merge dialog.

Mixed: Leave working tree untouched, reset index Resets the index but not the working tree (i.e., the changed files are preserved but not marked for commit) and reports what has not been updated. This is the git default action. This option can abort a merge.

Hard: Reset working tree and index (discard all local changes) Resets the index and working tree. Any changes to tracked files in the working tree since the selected commit are discarded. This option can abort a merge, and it is the default action in Abort Merge dialog.

Git hard reset does not use the Windows recycle bin

Unlike the revert or clean functions of TortoiseGit, the hard reset does not make use of the Windows recycle bin, i.e., uncommitted changes might get lost!

You can find more information at Section G.3.118, “git-reset(1)”.

2.25. Stash Changes

Often, when you’ve been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the git stash command.

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time (even on a different branch).

Tip

There is a similar function in TortoiseGit called changelists (Section 2.13, “Change Lists”) which can be used to structure commits.

When you want to record the current state of the working directory and the index, but want to go back to a clean working directory, right click on a folder to pop up the context menu and then select the command TortoiseGit → Stash changes A dialog will pop up where you can optionally enter a message for this state:

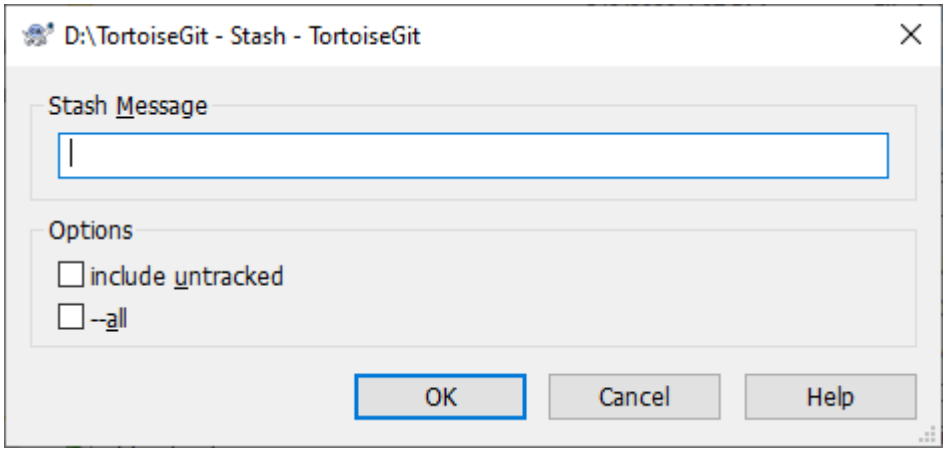


Figure 2.53. Stash changes dialog

You can also select include untracked, to stash untracked files away, too. To stash all files away, including ignored files in addition to the untracked files, select --all.

When TortoiseGit detects that a stashed changes exist, the context menu will be extended:

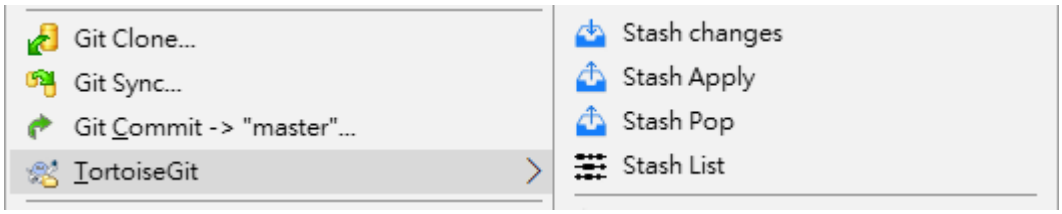


Figure 2.54. (un)stash options

The stash is implemented as a stack. Stash Apply will apply the changes of the latest stash to your working tree. Stash Pop does the same, but will remove the latest stash from the stack after applying it. Stash changes is still possible and will stash the current changes of the working copy to the top of the stack. Stash List provides an overview of all the whole stash stack. You can also remove and view the stashed changes there (similarly to the Section 2.14, “Log Dialog” and Section 2.16, “Reference Log”).

Conflicts

Although major merge work is done by git automatically applying a stash, a conflict may happen during cherry-picking (i.e., a file was modified in your current branch and also in the stash), please see Section 2.31, “Resolving Conflicts” on how to resolve conflicts.

Please note, that "REMOTE"/"theirs" in the conflict editor refers to the to be merged stash and "LOCAL"/"mine" to your version in the working tree before you applied the stash.

You can find more information at Section G.3.137, “git-stash(1)”.

2.26. Bisect

If you want to find out which revision introduced a bug, you can use the bisect functionality. Right click on a folder to pop up the context menu and then select the command TortoiseGit → Bisect start. A dialog will pop up:

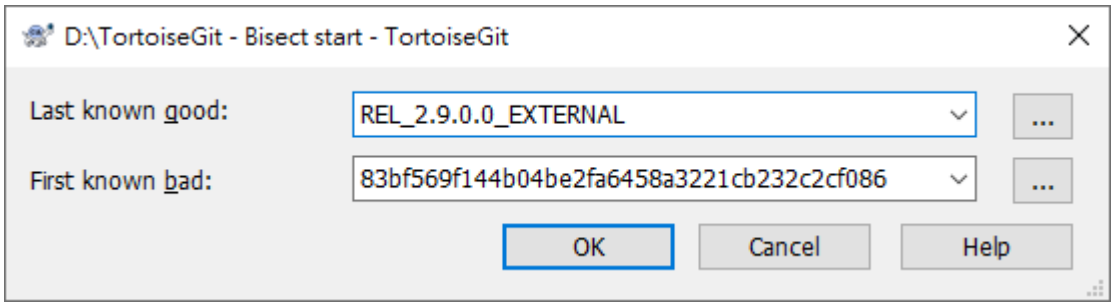


Figure 2.55. Bisect start

Enter the last known good revision and the first or one known bad (this is normally HEAD).

After hitting OK, Git will perform a binary search for the first faulty revision: Git switches to a revision in the middle. Now you can test this revision.

TortoiseGit now provides three new options in the context menu:

Figure 2.56. Bisect options

If this revision is OK, hit TortoiseGit → Bisect good , otherwise hit TortoiseGit → Bisect bad To skip this revision hit TortoiseGit → Bisect skip Git will proceed with the binary search and switches to the "next" revision, so that you can test it. This goes on until the faulty revision is found or you abort this operation by clicking on TortoiseGit → Bisect reset (this will reset the bisect process and switch out your previous branch/HEAD).

Selecting revisions

If a revision cannot be tested, or you want to go on with a different one, you can easily go to the log and (hard) reset the current HEAD to a revision you like.

Submodules

If you use submodule you might need to make sure that those are updated after each bisect step so that all dependencies are up to date.

You can find more information at Section G.3.8, “git-bisect(1)”

2.27. Branching/Tagging

One of the features of version control systems is the ability to isolate changes onto a separate line of development. This line is known as a *branch*. Branches are often used to try out new features without disturbing the main line of development with compiler errors and bugs. As soon as the new feature is stable enough then the development branch is *merged* back into the main branch.

Another feature of version control systems is the ability to mark particular revisions (e.g. a release version), so you can at any time recreate a certain build or environment. This process is known as *tagging*.

Git is very powerful at branching and tagging. It is very easy to create branches and tags.

2.27.1. Creating a Branch or Tag

Creating a branch is very simple: TortoiseGit → Create Branch...

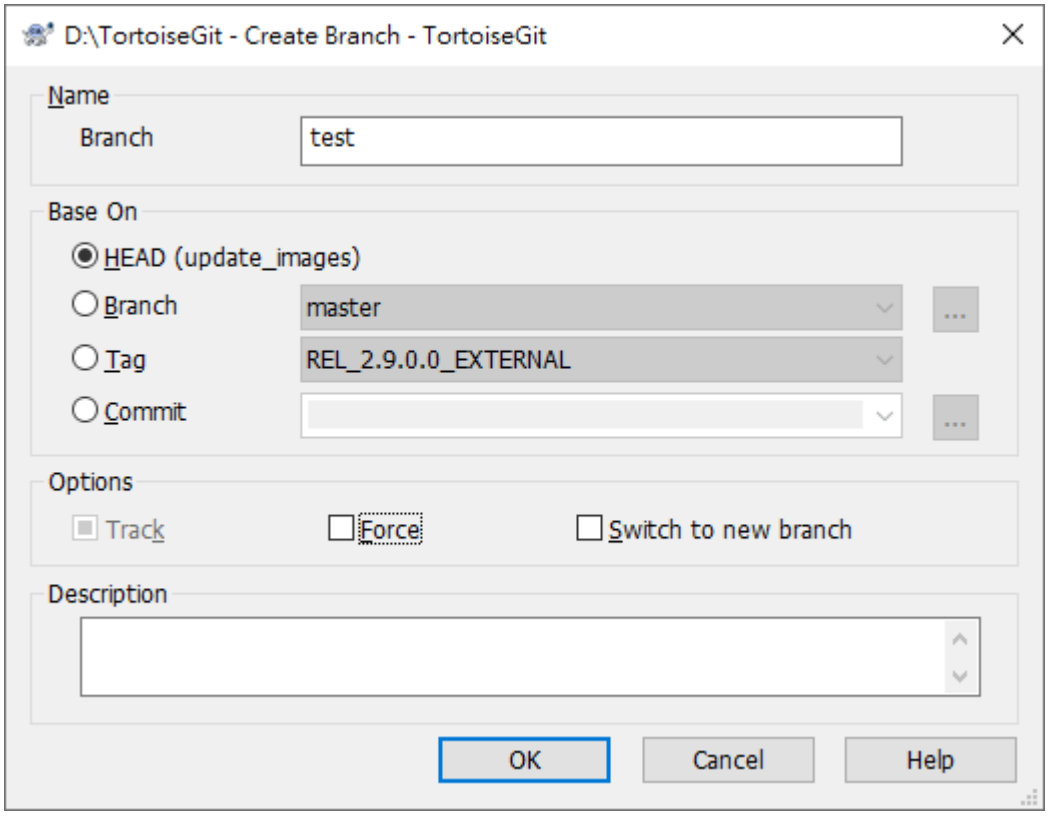


Figure 2.57. The Branch Dialog

Branch: input your branch name.

Creating a tag is very simple: TortoiseGit → Create Tag...

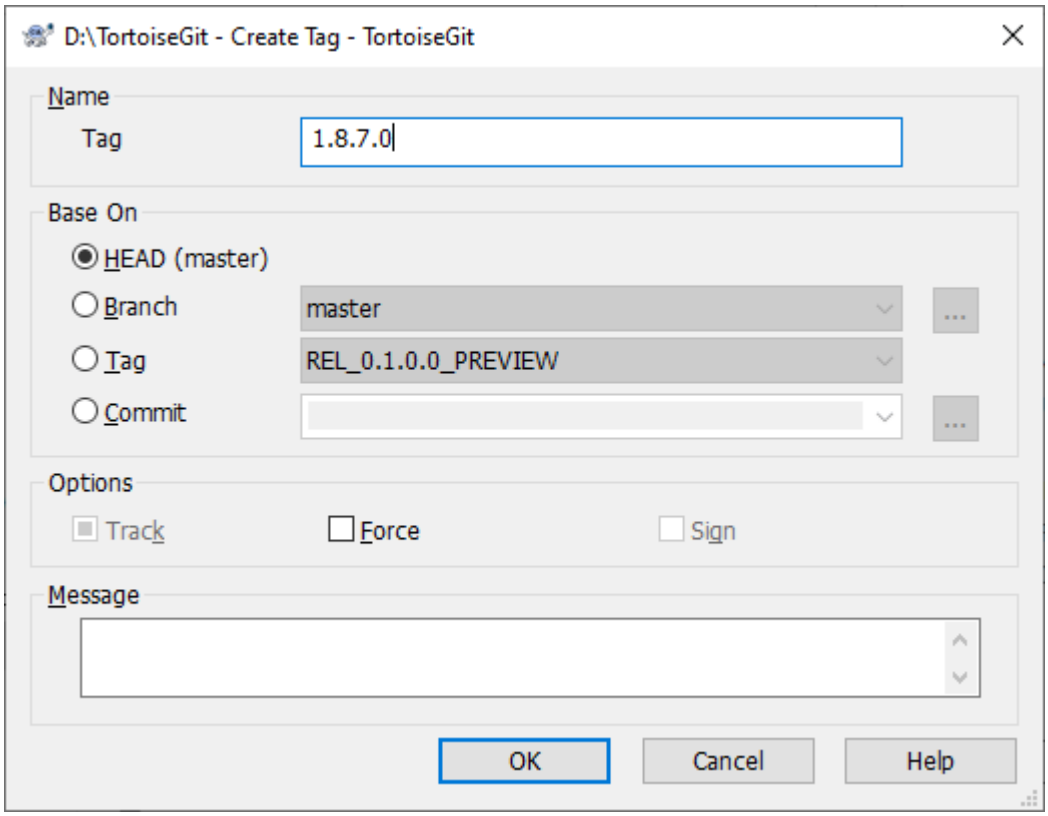


Figure 2.58. The Tag Dialog

Tag: input your tag name.

You can choose one commit that base on.

HEAD

Current commit checked out.

Branch

The latest commit of chosen branch.

Tag

The commit of chosen tag.

Commit

Any commit, you click ... to launch log dialog to choose commit. You also can input commit hash, or friendly commit name, such as HEAD~4.

If you want your working tree to be switched to the newly created branch automatically, use the Switch to new branch/tag checkbox. But if you do that, first make sure that your working tree does not contain modifications. If it does, those changes will be merged into the branch working tree when you switch.

track is a checkbox with three values. If it is checked `--track` is passed to git on OK, if it is unchecked `--no-track` is passed to git on OK. The third state indicates, that neither `--track` nor `--no-track` is passed to

git on OK - see `branch.autosetupmerge` configuration variable (Section G.3.29, “git-config(1)”) and `--track parameter` documentation for Section G.3.10, “git-branch(1)”.

Check Sign to create a GPG signed tag. This requires GPG and also the configuration variable `user.signingkey` to be set (see Section 2.36.6.2, “Git Config” and Section G.3.29, “git-config(1)”).

Tip

When using GPG 1.4 (which is shipped with Git for Windows) this requires a key *without* a passphrase. GPG >= 2 comes with an agent like pageant and, thus,.also works with passphrase protected keys, however, you might need to configure git to use the right `gpg.exe`. This can be done be setting the configuration variable `gpg.program` (e.g., `C:/Program Files (x86)/GNU/GnuPG/pub/gpg.exe`). We tested this with *Gpg4win* [https://www.gpg4win.de/] (Gpg4win vanilla is sufficient and with version 2.2.x it is also compatible to GPG 1.4 key files).

Press OK to create branch or tag at *local repository*.

Note that unless you opted to switch your working tree to the newly created branch, creating a Branch or Tag does *not* affect your working tree. Even if you create the branch from your working tree, those changes are committed to the original branch, not to the new branch.

On how to switch working tree to tag/branch, please refer to Section 2.4, “Checking Out A Working Tree (Switch to commit)”.

You can find more information at Section G.3.10, “git-branch(1)” and Section G.3.144, “git-tag(1)”.

2.28. Merging

Where branches are used to maintain separate lines of development, at some stage you will want to merge the changes made on one branch back into the other branch, or vice versa.

It is important to understand how branching and merging works in Git before you start using it, as it can become quite complex. For hints where to find more information about Git and merging see Section 2, “Reading Guide”.

The next point to note is that merging *always* takes place within a working tree. If you want to merge changes *into* a branch, you have to have a working tree for that branch checked out, and invoke the merge wizard from that working tree using TortoiseGit → Merge....

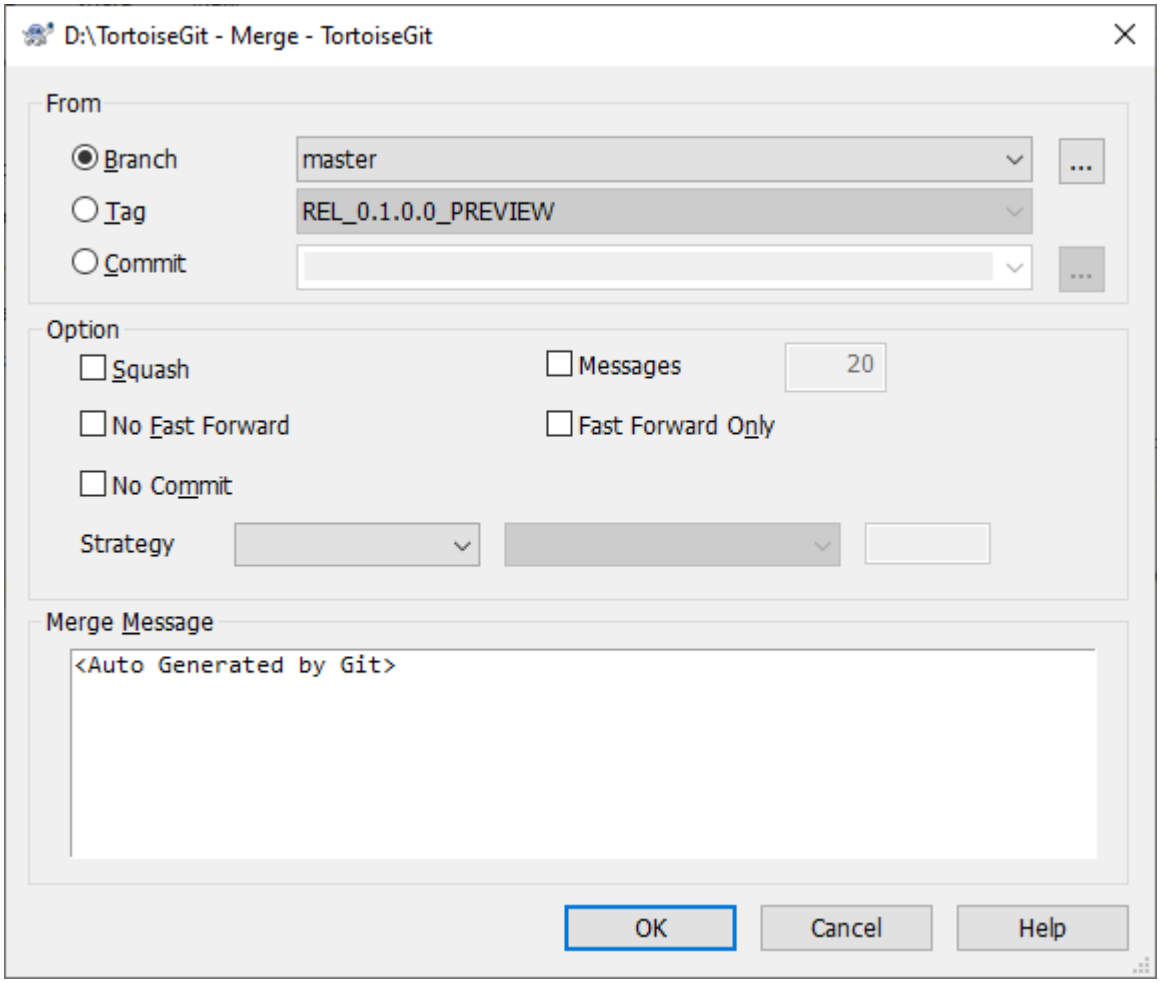


Figure 2.59. Merge dialog

In general it is a good idea to perform a merge into an unmodified working tree. If you have made other changes in your working tree, commit those first. If the merge does not go as you expect, you may want to abort the merge using the Abort Merge command which might discard *all* changes (depending on the mode, in case of hard).

You can choose one commit that you want to merge from.

HEAD

Current commit checked out.

Branch

The latest commit of chosen branch.

Tag

The commit of chosen tag.

Commit

Any commit, you click ... to launch log dialog to choose commit. You also can input commit hash, or friendly commit name, such as HEAD~4.

Squash Just merge change from the other branch. Can't recorder Merge information. The new commit will not record merge branch as one parent commit. Log view will not show merge line between two branch.

No Fast Forward Generate a merge commit even if the merge resolved as a fast-forward. See https://stackoverflow.com/q/41794529/3906760 for an example of fast-forward vs. non-fast-forward merge.

No Commit Do not automatically create a commit after merge.

Messages Populate the log message with one-line descriptions from the actual commits that are being merged. Can specify the number of commits to be included in the merge message.

Conflicts

Although major merge work is done by git automatically, a conflict may happen during merge (i.e., a file is modified in both branches, the current one and the one you want to merge), please see Section 2.31, “Resolving Conflicts” on how to resolve conflicts.

Please note, that "REMOTE"/"theirs" in the conflict editor refers to the changes your on the branch you selected for merging and "LOCAL"/"mine" to your HEAD version in your working tree.

You can see more information at Section G.3.86, “git-merge(1)”.

2.29. Cherry picking

Cherry-picking in TortoiseGit is invoked from the Revision Log Dialog. Within this dialog, select the commit(s) to cherry-pick, then right-click on one of the selected commits to pop up the context menu. Select Cherry Pick this commit... (or Cherry Pick select commits... if more than one commit is selected).

The Cherry Pick dialog will be shown.

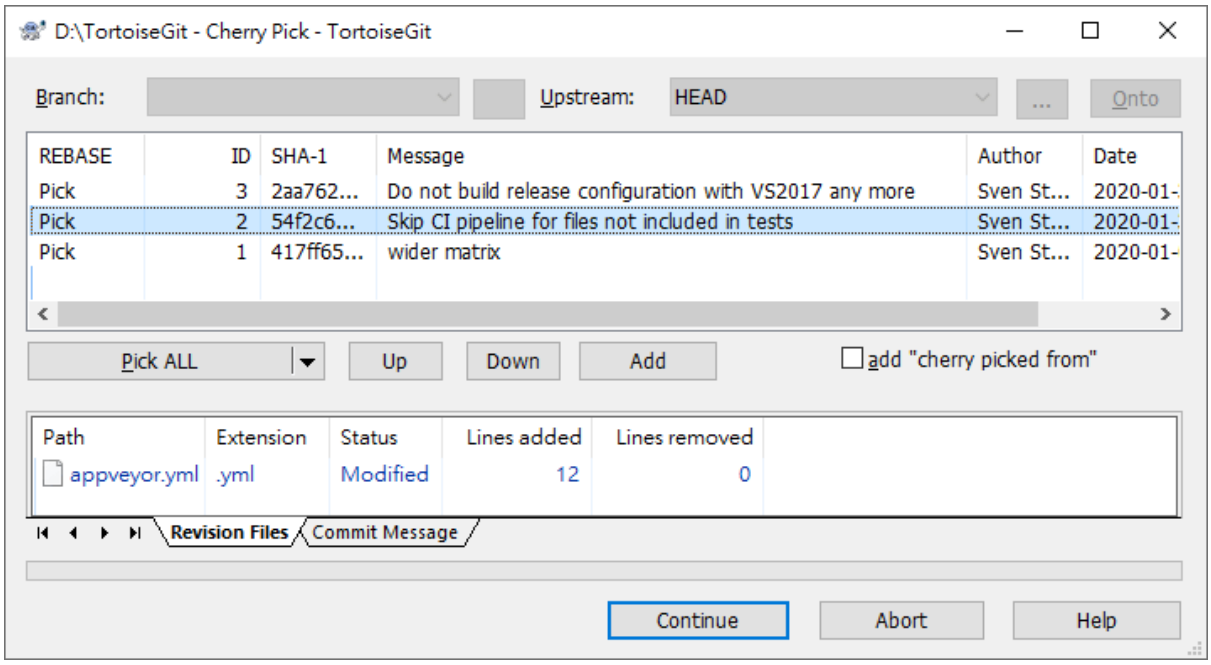


Figure 2.60. Cherry Pick dialog

The Cherry Pick dialog is similar to the Rebase dialog. The top table displays one line for each selected commit to cherry-pick. Buttons below it control the actions (Pick, Squash, Edit, Skip) and the order in which multiple commits are picked. Selecting a line shows the files affected by the commit.

Conflicts

Although major merge work is done by git automatically while cherry-picking, a conflict may happen during cherry-picking (i.e., a file was modified in your current branch and also in one or more commits you are cherry-picking), please see Section 2.31, “Resolving Conflicts” on how to resolve conflicts.

> Please note, that "REMOTE"/"theirs" in the conflict editor refers to the changes your are picking and "LOCAL"/"mine" to your HEAD version in your working tree.

You can find more information at Section G.3.20, “git-cherry-pick(1)”.

2.30. Rebase

Rebase is quite complex and it alters/rewrites the history of a repository. Please make sure you understood its principles before using it (for general hints where to find more information about Git and rebasing see Section 2, “Reading Guide” and especially Section G.3.107, “git-rebase(1)”).

TortoiseGit → Rebase

The Rebase dialog will be shown.

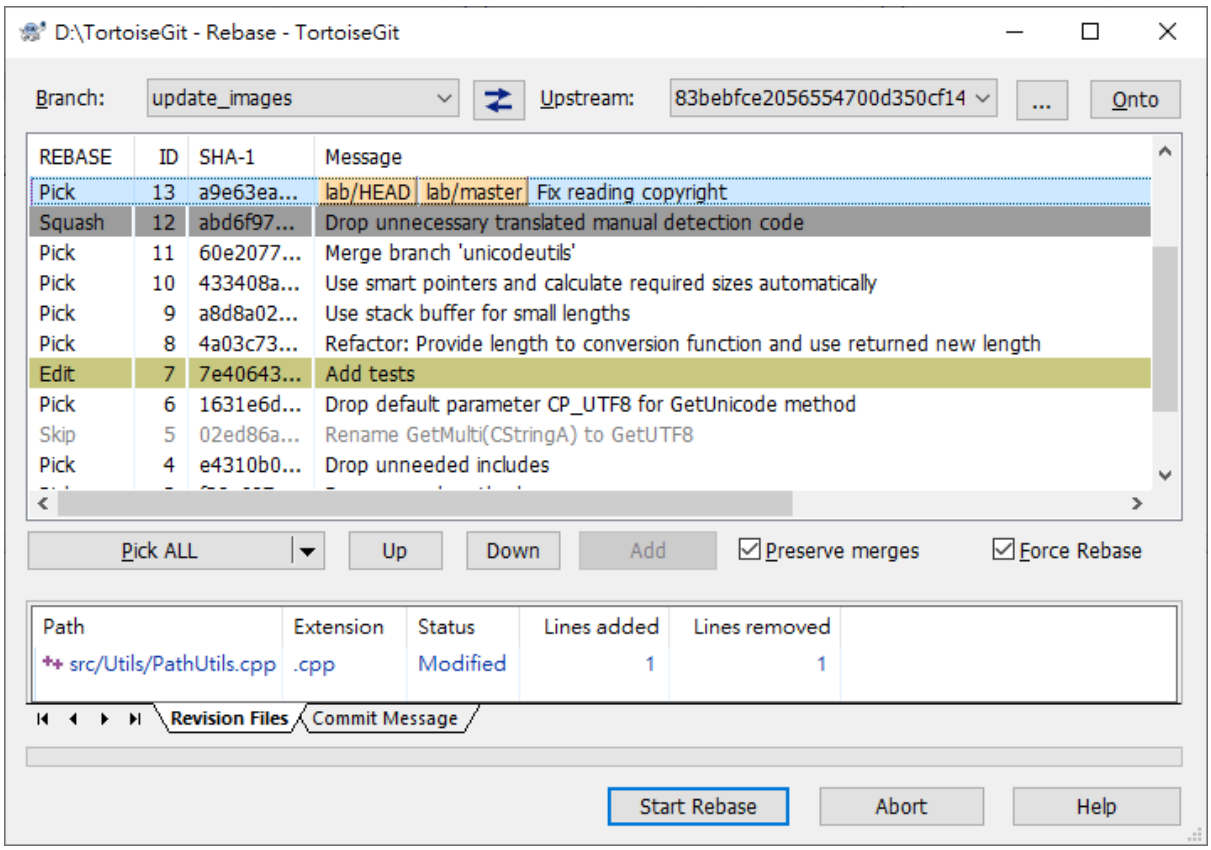


Figure 2.61. Rebase dialog

Rebasing commits takes places from the bottom of the list to the top (in ascending order of the ID column). For example, `squash` means that the commit gets squashed/combined with the previous commit(s) which are located below in the list (with a lower ID).

Tip

Instead of setting `pick`, `skip`, `edit`, `squash` by using the context-menu, you can also use the following keys: **space**: shifts the state, **s**: `skip`, **e**: `edit`, **p**: `pick`, **q**: `squash`

Tip

There is a button that swaps branch and upstream. Assume you are currently working on `master` branch, and wish to rebase `feature` branch onto `master`. Instead of switching to `feature` in

advance, select the commit of `feature` in log list, Context Menu → Rebase and click this swap button. TortoiseGit's rebase moves `feature` to `master` directly, then cherry-picks the commits. This approach touches fewer files and runs faster.

Important

When preserving merge commits, re-ordering commits cannot be handled properly in all cases, see in known bugs of vanilla git rebase: Section G.3.107, “git-rebase(1)”.

Conflicts

Although major merge work is done by git automatically while rebasing, a conflict may happen during rebase (i.e., a file was modified in both branches, the one you are rebasing one and the on which you are rebasing), please see Section 2.31, “Resolving Conflicts” on how to resolve conflicts.

Please note, that "REMOTE"/"theirs" in the conflict editor refers to the changes of the branch you rebase onto and "LOCAL"/"mine" to your version on the branch which you are rebasing.

2.31. Resolving Conflicts

During a merge, the working tree files are updated to reflect the result of the merge. Once in a while, you will get a *conflict* when you merge another branch, cherry-pick commits, rebase or apply a stash: Among the changes made to the common ancestor's version, non-overlapping ones (that is, you changed an area of the file while the other side left that area intact, or vice versa) are incorporated in the final result verbatim. When both sides made changes to the same area, however, Git cannot randomly pick one side over the other, and asks you to resolve it by leaving what both sides did to that area. Whenever a conflict is reported you need to resolve it!

The conflicting area is marked in the file like this (also cf. the section called “HOW CONFLICTS ARE PRESENTED”):

```
<<<<<<< yours
    your changes
=======
    changes from the code merged
>>>>>>> their
```

You can use any editor to manually resolve the conflict or you can launch an external merge tool/conflict editor with TortoiseGit → Edit Conflicts. Then TortoiseGit will place three additional files in your directory for the selected conflicted file and launch the configured conflict editor:

filename.ext.BASE.ext

This is the common ancestor's version of the conflicted file (this version does contain neither any of your nor any of the changes of the to be merged branch/revision, especially it does not contain any conflict markers).

filename.ext.LOCAL.ext

This is your file as it existed in your working tree before you started the merge (i.e., the file conforms to the latest committed state of the HEAD of your local repository) - that is, without conflict markers. Therefore, this state/version is often also called "mine".

Just for completeness "mine" means for "stash"/"merge"/"pull"/"cherry-pick" the HEAD version in your working tree and for "rebase" the version on the branch you rebase.

filename.ext.REMOTE.ext

This is the version of file of the revision you want to merge (on a normal merge this correspondents to MERGE_HEAD). As you want to merge other changes, this state/version is often also called "theirs".

Just for completeness "theirs" means for "stash"/"merge"/"pull"/"cherry-pick" the version of the to be merged commit/branch and for "rebase" the version of the branch you rebase onto.

Afterwards execute the command TortoiseGit → Resolved and commit your modifications to the repository (if the conflict occurred while rebasing or cherry-picking make sure you use the cherry-pick resp. rebase dialog for committing and not the normal commit dialog!). Please note that the Resolve command does not really resolve the conflict. It uses "git add" to mark file status as resolved to allow you to commit your changes and it removes the `filename.ext.BASE.ext`, `filename.ext.LOCAL.ext` and `filename.ext.REMOTE.ext` files.

If you have conflicts with binary files, Git does not attempt to merge the files itself. The local file remains unchanged (exactly as you last changed it). In order to resolve the conflict use TortoiseGit → Resolve... and then right click on the conflicted file and choose one of Resolved (the current version of the file which is in the working tree will be used), Resolve conflict using 'mine' (the version of the file of your HEAD will be used), and Resolve conflict using 'theirs' (the version of the file of the merged revision/branch will be used). After that commit.

You can use the Resolved command for multiple files if you right click on the parent folder and select TortoiseGit → Resolve... This will bring up a dialog listing all conflicted files in that folder, and you can select which ones to mark as resolved.

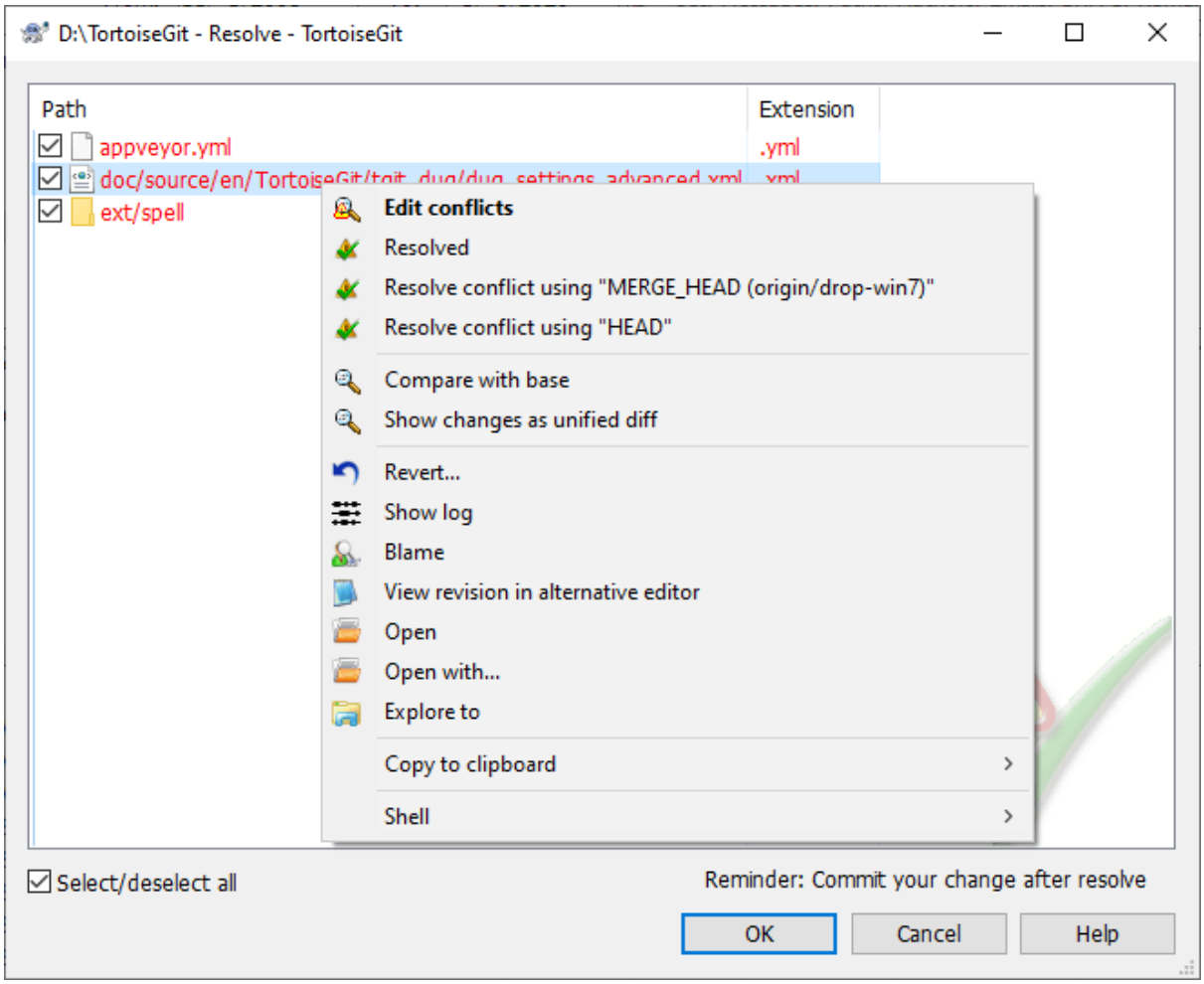


Figure 2.62. The resolve conflicts dialog

Important

Git (unlike SVN) does not automatically create `filename.ext.BASE.ext`, `filename.ext.LOCAL.ext` and `filename.ext.REMOTE.ext` files for conflicted files. These are only created on-demand by TortoiseGit when you use the command Edit Conflicts.

Important

In Git (unlike SVN) you have to commit after resolving conflicts. However, if the conflict occurred while rebasing or cherry-picking make sure you use the cherry-pick resp. rebase dialog for committing and not the normal commit dialog!

2.31.1. Special conflict cases

2.31.1.1. Delete-modify conflicts

A special conflict case is a delete-modify conflict. Here, a file is deleted on one branch and the same file is modified on another branch. In order to resolve this conflict the user has to decide whether to keep the modified version or delete the file from the working tree.

Figure 2.63. Resolve delete-modify conflict Dialog

2.31.1.2. Submodule conflicts

Another special conflict case is a conflict involving a submodule. Here, a submodule is changed in different (conflicting) ways on two branches.

The resolve submodule conflict dialog shows the base, the local and the remote commit of the conflicting submodule as well as the commit type (rewind, fast-forward, ...).

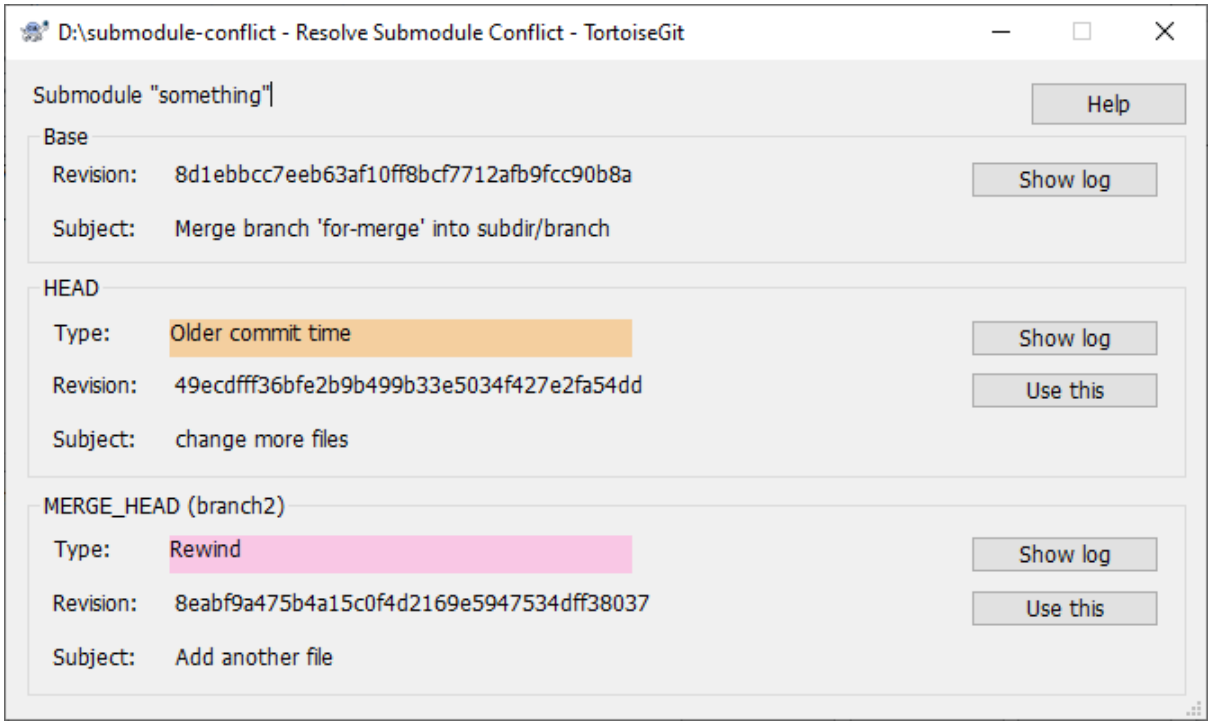


Figure 2.64. Resolve submodule conflict Dialog

Uninitialized submodules

If the submodule is not yet initialized the resolve submodule conflict dialog only shows the commit IDs (SHA-1). Also, the conflict cannot be resolved automatically: First, you have to manually clone the submodule into the right folder. Then, you can resolve the conflict using TortoiseGit or git (by checking out the right commit in the submodule and committing the parent working tree).

2.32. Creating and Applying Patches and Pull Requests

For open source projects (like this one) everyone has read access to the (main/public) repository, and anyone can make a contribution to the project. So how are those contributions controlled? If just anyone could commit changes this central repository, the project would be permanently unstable and probably permanently broken. In this situation the change is managed by submitting a *patch* file or a *pull request* to the development team, who do have write access. They can review the changes first, and then either submit it to the main repository or reject it back to the author.

Patch files are simply Unified-Diff files showing the differences between your working tree and the base revision.

A pull request is an request to another repository owner to pull changes from your repository. I.e. you must have access to a public repository where you can push your changes (normally a special branch).

2.32.1. Creating a Patch Serial

First you need to make *and test* your changes. Then you commit your changes via TortoiseGit → Commit... on the parent folder, enter a good commit message. After that select TortoiseGit → Create Patch Serial... and choose the correct options to include your changes/commits.

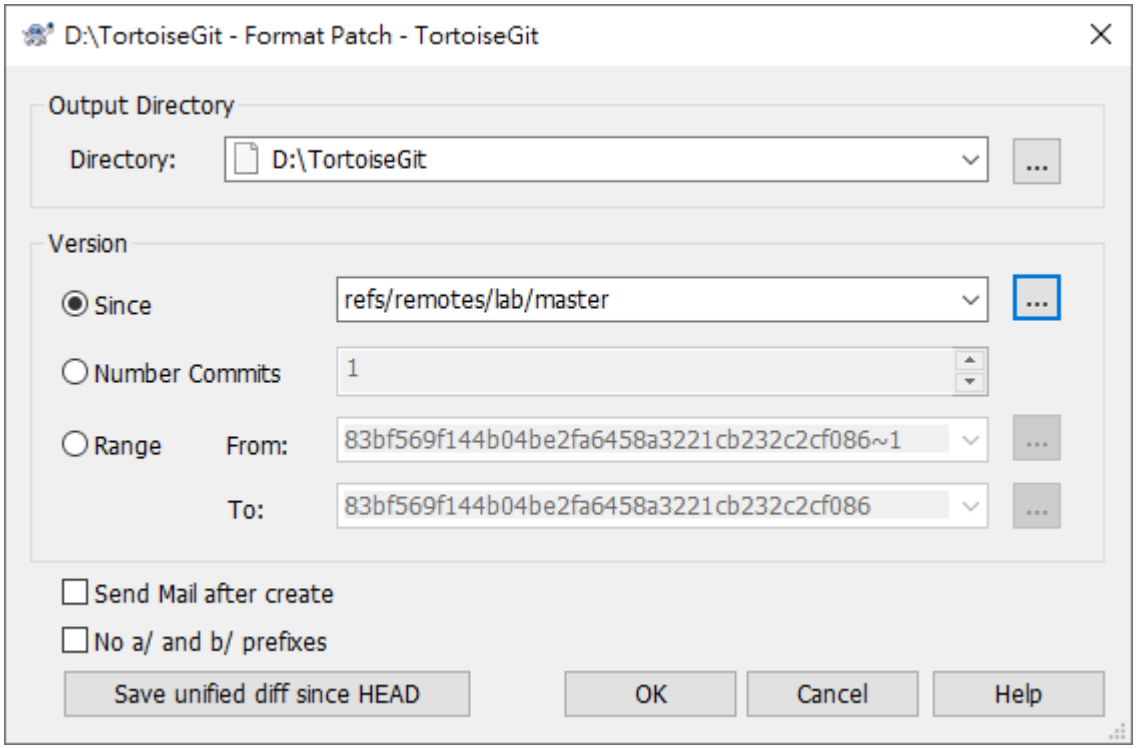


Figure 2.65. The Create Patch dialog

Directory is output directory of patch. Patch file name will be created by commit subject.

Since create patch from point. You can click ... to launch reference browser dialog to choose branch or tag (cf. Section 2.11, "Browse All Refs").

Number Commits is limited how much patch will created.

Range is choose range of from commit to to. You can click ... to launch log dialog to choose commit.

Send Mail after create launch send mail dialog after patches created (see Section 2.32.2, "Sending patches by mail").

You can find more information at Section G.3.54, "git-format-patch(1)".

Important

Here Git is different to TortoiseSVN: In TortoiseSVN you directly create a patch instead of committing your changes and create a patch of the commits afterwards (in git you have a full local copy/fork of the project you cloned - commits are just local). To generate a patch containing the uncommitted, but staged, changes click on Save unified diff since HEAD.

For hints where to find more information about doing version control with Git see Section 2, "Reading Guide".

2.32.2. Sending patches by mail

In order to send patches to the upstream authors, select the patch files and then right click on them and select TortoiseGit → Send Mail...

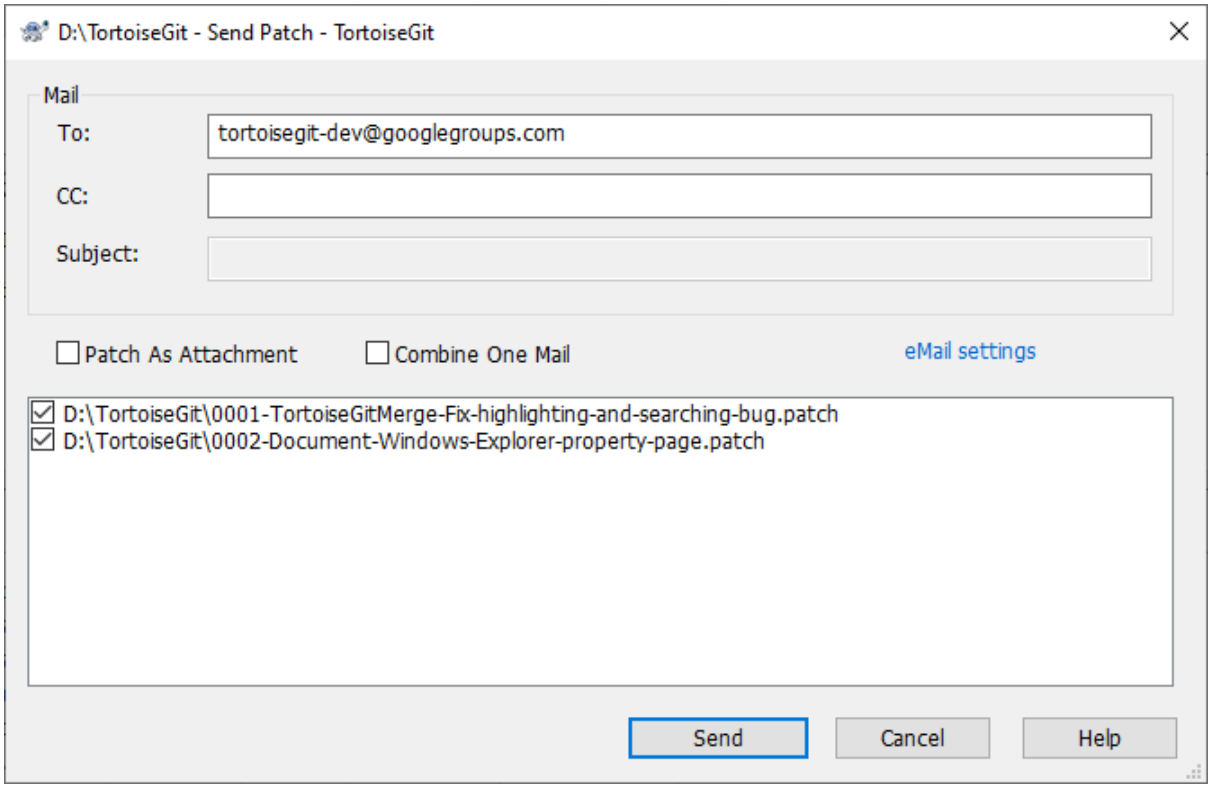


Figure 2.66. The Send Patches Dialog

First you need to enter the recipient(s) (To and/or CC).

Depending on the mail type (Patch as attachment or Combine One Mail) you have to enter a Subject for the mail.

Patch as attachment adds the patch(es) as attachment(s) to the mail(s) instead of inlining them.

Combine One Mail adds all patches to one mail. You have to enter a Subject for the mail in this case.

2.32.3. Applying a single Patch File

Patch files are applied to your working tree. This should be done from the same folder level as was used to create the patch. If you are not sure what this is, just look at the first line of the patch file. For example, if the first file being worked on was `doc/source/english/chapter1.xml` and the first line in the patch file is `Index: english/chapter1.xml` then you need to apply the patch to the `doc/source/` folder. However, provided you are in the correct working tree, if you pick the wrong folder level, TortoiseGit will notice and suggest the correct level.

From the context menu for a patch file (`.patch` or `.diff` extension), click on TortoiseGit → Review/apply single patch... You might be prompted to enter a working tree location:

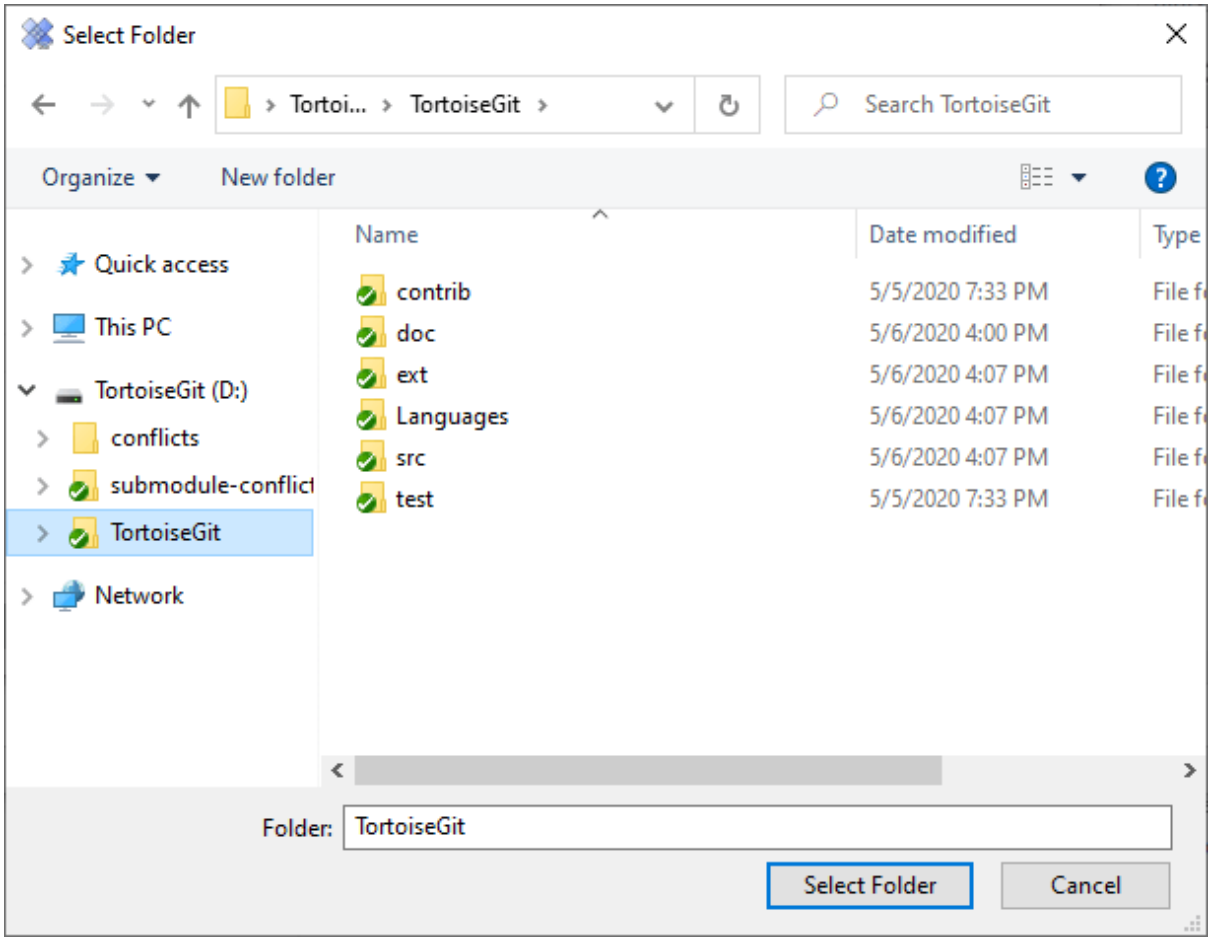


Figure 2.67. The Choose Repository Dialog

If the working tree is found, TortoiseGitMerge is launched to show and apply differences.

2.32.4. Applying a Patch Serial

Patch files are applied to your working tree. For this copy the patch (or mbox) files to the root of your working tree.

From the context menu for that folder (or all marked patch files), click on TortoiseGit → Apply Patch Serial...

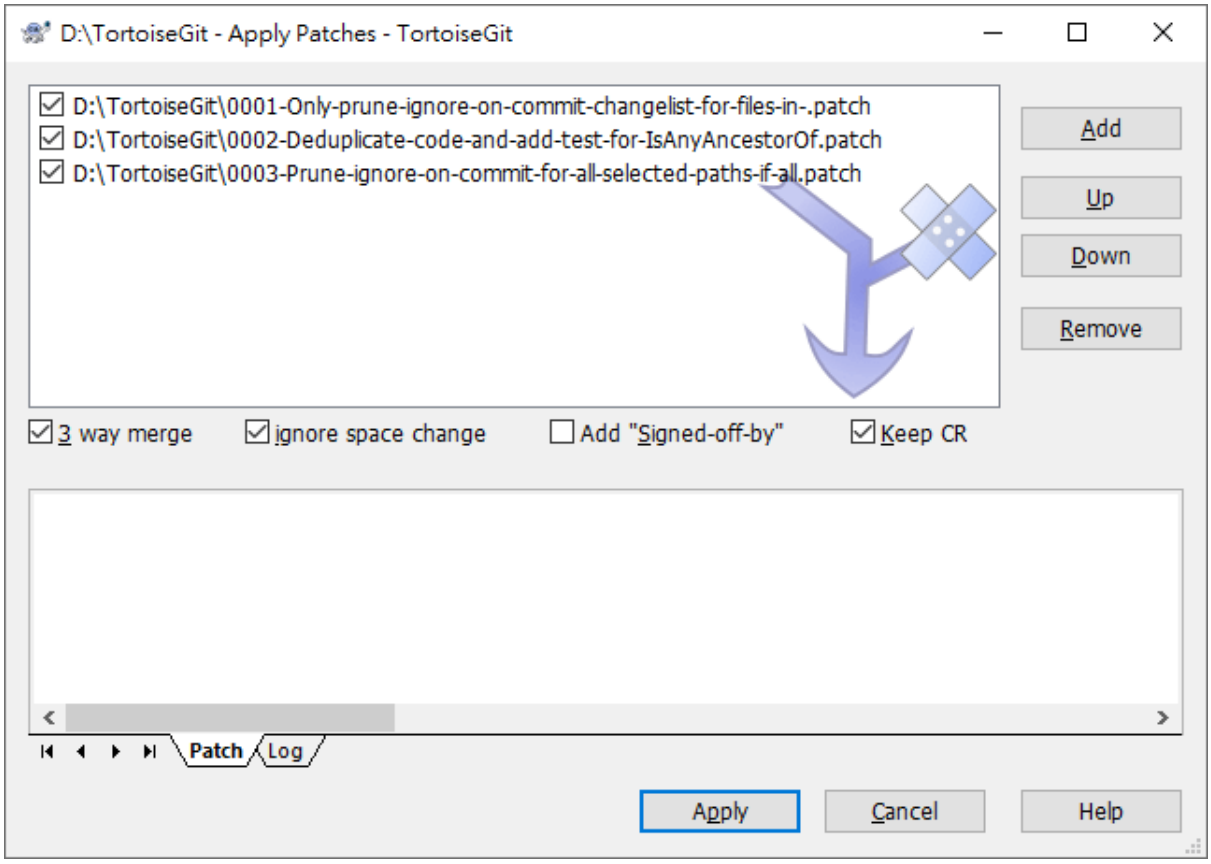


Figure 2.68. The Apply Patch Dialog

Add Insert patch

Up Move chosen patch up.

Down Move chosen patch down.

Remove Remove the chosen patch.

Apply Start applying the patches one by one.

You can find more information at Section G.3.3, “git-am(1)”.

2.32.5. Creating a pull request

Apart from sending patches (or patch serials) to other developers, there are two ways to ask other people to integrate your changes into their repositories.

First: After pushing your changes to a (public) repository, you just provide other people the URL of your repository and the name of the branch or the revision id. E.g.: `git://example.com/repo.git BRANCHNAME`

Second: After pushing your changes to a (public) repository, you can create a standardized (quite formal) request for other people to pull your changes and integrate them into their repository. The format pull request consists of a list of all commits and provides some statistics about changed files, so that other people can a quick overview.

Select Request pull on the progress dialog after pushing your changes.

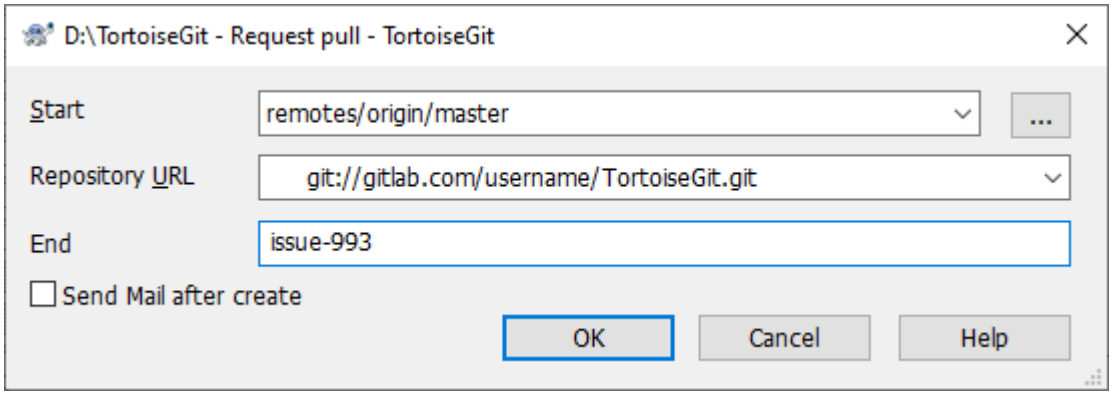


Figure 2.69. The Request Pull Dialog

Start

This should be the revision on which your changes are based on.

URL

The public URL to your repository, which can be access by the people who shall pull your changes.

End

This should be the branch name or revision id of the end of your commits.

After clicking on OK the pull request is created. Just copy it and pass it to other people who you want to pull your changes.

You can find more information at Section G.3.116, “git-request-pull(1)”.

2.33. Who Changed Which Line?

Sometimes you need to know not only what lines have changed, but also who exactly changed specific lines in a file. That's when the TortoiseGit → Blame... command, sometimes also referred to as *annotate* command comes in handy.

This command lists, for every line in a file, the author and the revision the line was changed.

2.33.1. Blame for Files

By default the blame file is viewed using *TortoiseGitBlame*, which highlights the different revisions to make it easier to read.

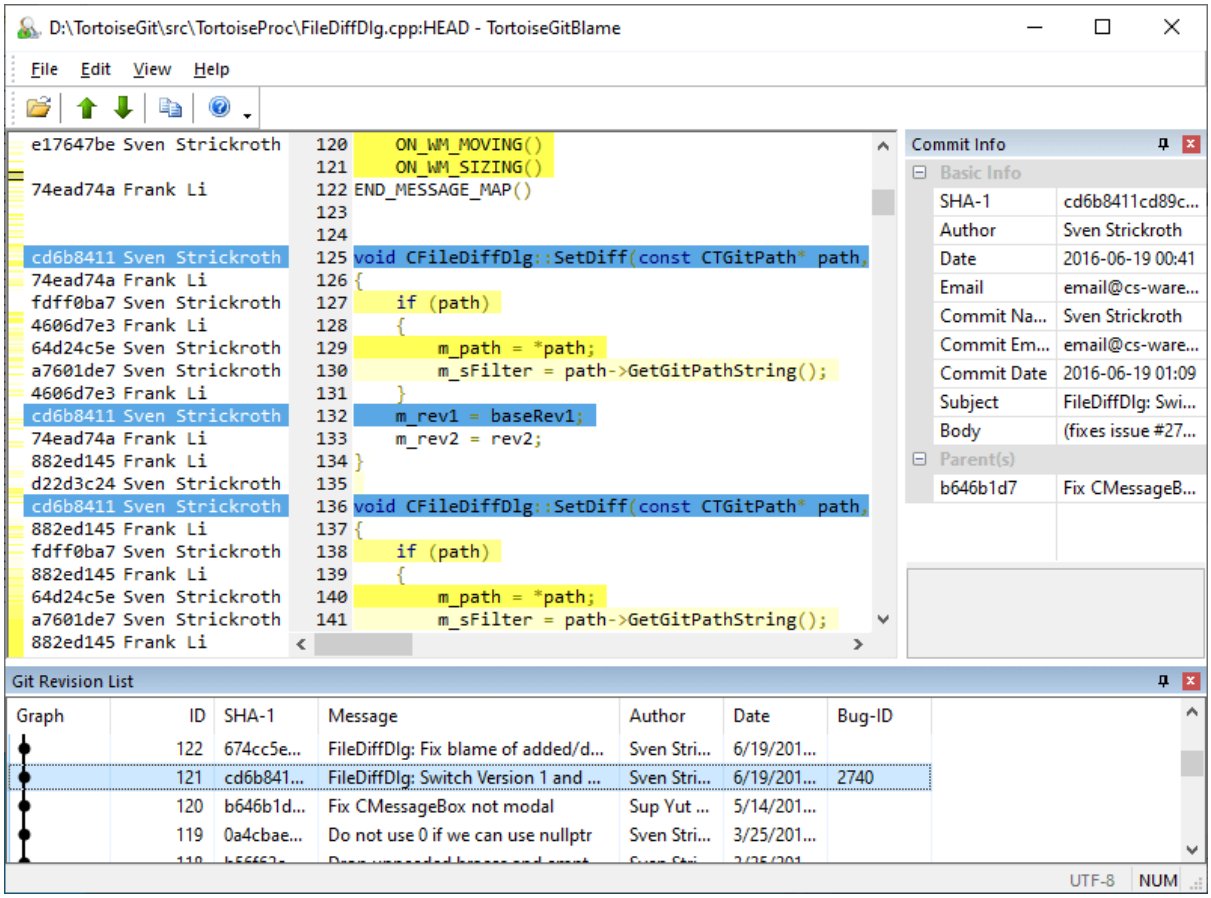


Figure 2.70. TortoiseGitBlame

TortoiseGitBlame, which is included with TortoiseGit. When you hover the mouse over a line in the blame info column, all lines with the same revision are shown with a darker background. Lines from other revisions which were changed by the same author are shown with a light background. The coloring may not work as clearly if you have your display set to 256 color mode.

If you left click on a line (on the blame info column on the left), all lines with the same revision are highlighted, and lines from other revisions by the same author are highlighted in a lighter color. This highlighting is sticky, allowing you to move the mouse without losing the highlights. Click on that revision again to turn off highlighting.

The revision comments (log message) are shown in a hint box whenever the mouse hovers over the blame info column. If you want to copy the log message for that revision, use the context menu which appears when you right click on the blame info column.

If you need a better visual indicator of where the oldest and newest changes are, select View → Colorize by age, continuous. Then the background color intensity of the lines is related to its age. This will use a color gradient to show newer lines in yellow and older lines in white. The default coloring is quite light, but you can change it using the TortoiseGitBlame settings.

Please also check out the View menu. There you can toggle the Ignore whitespace and also toggle the detection of moved/copied lines from other files and Follow renames.

You can search within the Blame report using Edit → Find.... This allows you to search for revision numbers, authors and the content of the file itself. Log messages are not included in the search - you should use the Log Dialog to search those.

You can also jump to a specific line number using Edit → Go To Line....

When the mouse is over the blame info columns, a context menu is available which helps with comparing revisions and examining history, using the commit of the line under the mouse as a reference. Context menu → Blame

previous revision generates a blame report for the same file, but using the previous revision as the upper limit. This gives you the blame report for the state of the file just before the line you are looking at was last changed. Context menu → Show changes starts your diff viewer, showing you what changed in the referenced revision of the file. Please note, however, that these two options are only available if this line is not there since the initial commit of the file. Context menu → Show log displays the revision log dialog starting with the referenced revision.

The settings for TortoiseGitBlame can be accessed using TortoiseGit → Settings... on the TortoiseGitBlame tab. Refer to Section 2.36.8, “TortoiseGitBlame Settings”.

You can find more information at Section G.3.9, “git-blame(1)”.

2.34. Exporting a Git Working Tree

Sometimes you may want a snapshot of a specific revision/commit, e.g. to create a zipped tarball of your source, or to export to a web server. for this TortoiseGit offers the command TortoiseGit → Export....

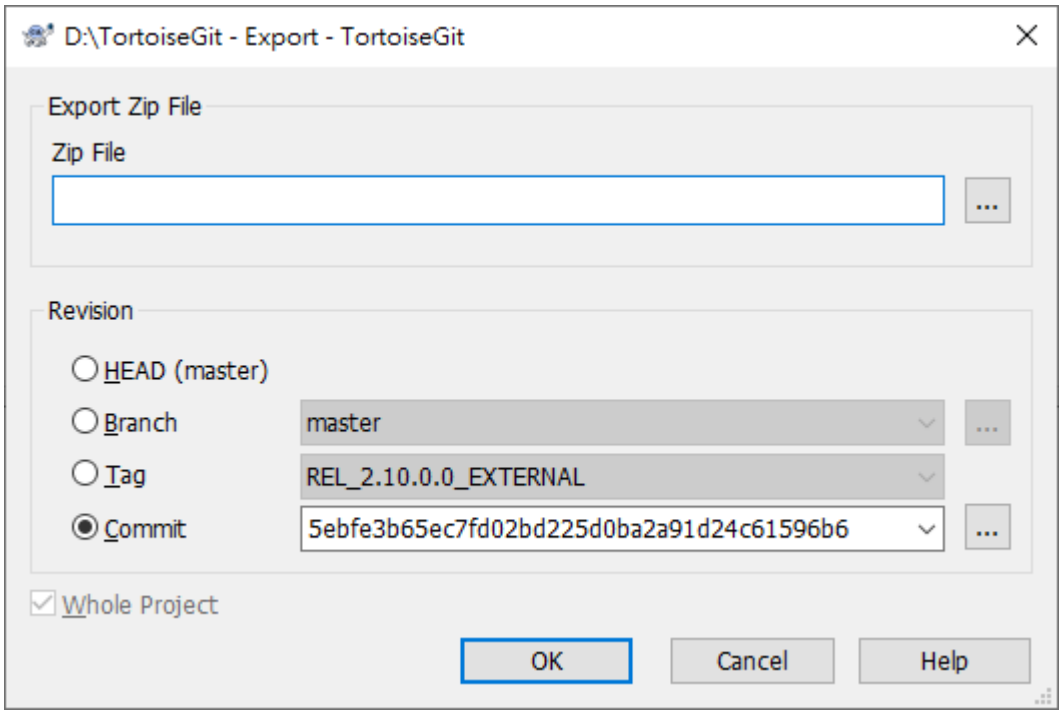


Figure 2.71. The Export Dialog

Zip File zip file of export

HEAD

Current commit checked out.

Branch

The latest commit of chosen branch.

Tag

The commit of chosen tag.

Commit

Any commit, you click ... to launch log dialog to choose commit. You also can input commit hash, or friendly commit name, such as `HEAD~4`.

You can find more information at Section G.3.7, “git-archive(1)”.

Exporting single files

The export dialog does not allow exporting single files.

To export single files with TortoiseGit, you have to use the repository browser (cf. Section 2.17, “The Repository Browser”) or log dialog (cf. Section 2.14, “Log Dialog”). Simply drag the file(s) you want to export from the repository browser to where you want them in the explorer, or use the context menu in the repository browser to export the files.

2.35. Integration with Bug Tracking Systems / Issue Trackers

It is very common in Software Development for changes to be related to a specific bug or issue ID. Users of bug tracking systems (issue trackers) would like to associate the changes they make in Git with a specific ID in their issue tracker. Most issue trackers therefore provide a pre-commit hook script which parses the log message to find the bug ID with which the commit is associated. This is somewhat error prone since it relies on the user to write the log message properly so that the pre-commit hook script can parse it correctly.

TortoiseGit can help the user in two ways:

1. When the user enters a log message, a well defined line including the issue number associated with the commit can be added automatically. This reduces the risk that the user enters the issue number in a way the bug tracking tools can't parse correctly.

   Or TortoiseGit can highlight the part of the entered log message which is recognized by the issue tracker. That way the user knows that the log message can be parsed correctly.

2. When the user browses the log messages, TortoiseGit creates a link out of each bug ID in the log message which fires up the browser to the issue mentioned.

2.35.1. Adding Issue Numbers to Log Messages

You can integrate a bug tracking tool of your choice in TortoiseGit. To do this, you have to define some configuration, which start with `bugtraq.`. These settings can be edited using TortoiseGit settings dialog: Section 2.36.7.2, “Config”

There are two ways to integrate TortoiseGit with issue trackers. One is based on simple strings, the other is based on *regular expressions*. The configuration used by both approaches are:

bugtraq.url

Set this configuration to the URL of your bug tracking tool. It must be properly URI encoded and it has to contain `%BUGID%`. `%BUGID%` is replaced with the Issue number you entered. This allows TortoiseGit to display a link in the log dialog, so when you are looking at the revision log you can jump directly to your bug tracking tool. You do not have to provide this configuration, but then TortoiseGit shows only the issue number and not the link to it. e.g the TortoiseGit project is using `https://tortoisegit.org/issue/%BUGID%`

bugtraq.warnifnoissue

Set this to `true`, if you want TortoiseGit to warn you because of an empty issue-number text field. Valid values are `true/false`. *If not defined, `false` is assumed.*

2.35.1.1. Issue Number in Text Box

In the simple approach, TortoiseGit shows the user a separate input field where a bug ID can be entered. Then a separate line is appended/prepended to the log message the user entered.

bugtraq.message

This configuration activates the bug tracking system in *Input field* mode. If this configuration is set, then TortoiseGit will prompt you to enter an issue number when you commit your changes. It's used to add a line at the end of the log message. It must contain `%BUGID%`, which is replaced with the issue number on commit. This ensures that your commit log contains a reference to the issue number which is always in a consistent format and can be parsed by your bug tracking tool to associate the issue number with a particular commit. As an example you might use `Issue : %BUGID%`, but this depends on your Tool.

bugtraq.append

This configuration defines if the bug-ID is appended (true) to the end of the log message or inserted (false) at the start of the log message. Valid values are `true/false`. *If not defined, `true` is assumed, so that existing projects don't break.*

bugtraq.label

This text is shown by TortoiseGit on the commit dialog to label the edit box where you enter the issue number. If it's not set, `Bug-ID / Issue-Nr:` will be displayed. Keep in mind though that the window will not be resized to fit this label, so keep the size of the label below 20-25 characters.

bugtraq.number

If set to `true` only numbers are allowed in the issue-number text field. An exception is the comma, so you can comma separate several numbers. Valid values are `true/false`. *If not defined, `true` is assumed.*

2.35.1.2. Issue Numbers Using Regular Expressions

In the approach with *regular expressions*, TortoiseGit doesn't show a separate input field but marks the part of the log message the user enters which is recognized by the issue tracker. This is done while the user writes the log message. This also means that the bug ID can be anywhere inside a log message! This method is much more flexible, and is the one used by the TortoiseGit project itself.

bugtraq.logregex

This configuration activates the bug tracking system in *Regex* mode. It contains either a single regular expressions, or two regular expressions separated by a newline.

If two expressions are set, then the first expression is used as a pre-filter to find expressions which contain bug IDs. The second expression then extracts the bare bug IDs from the result of the first regex. This allows you to use a list of bug IDs and natural language expressions if you wish. e.g. you might fix several bugs and include a string something like this: “This change resolves issues #23, #24 and #25”

If you want to catch bug IDs as used in the expression above inside a log message, you could use the following regex strings, which are the ones used by the TortoiseGit project: `[Ii]ssues?:?(\s*(,|and)?\s*#\d+)+` and `(\d+)`

The first expression picks out “issues #23, #24 and #25” from the surrounding log message. The second regex extracts plain decimal numbers from the output of the first regex, so it will return “23”, “24” and “25” to use as bug IDs.

Breaking the first regex down a little, it must start with the word “issue”, possibly capitalised. This is optionally followed by an “s” (more than one issue) and optionally a colon. This is followed by one or more groups each having zero or more leading whitespace, an optional comma or “and” and more optional space. Finally there is a mandatory “#” and a mandatory decimal number.

If only one expression is set, then the bare bug IDs must be matched in the groups of the regex string. Example: `[Ii]ssue(?:s)? #?(\d+)` This method is required by a few issue trackers, e.g. trac, but it is harder to construct the regex. We recommend that you only use this method if your issue tracker documentation tells you to.

If you are unfamiliar with regular expressions, take a look at the introduction at *https://en.wikipedia.org/wiki/Regular_expression* [https://en.wikipedia.org/wiki/Regular_expression], and the online documentation and tutorial at *https://www.regular-expressions.info/* [https://www.regular-expressions.info/].

If both the `bugtraq:message` and `bugtraq:logregex` properties are set, `logregex` takes precedence.

Tip

Even if you don't have an issue tracker with a pre-commit hook parsing your log messages, you still can use this to turn the issues mentioned in your log messages into links!

And even if you don't need the links, the issue numbers show up as a separate column in the log dialog, making it easier to find the changes which relate to a particular issue.

2.35.1.3. Issue Tracker Provider Settings based on Hierarchical Git Configuration

This is a hierarchical git configuration to associate issue tracker plugin with your project, rather than with to a specific directory path. Such settings are more portable. To deploy the settings, set to Project level and commit `.tgitconfig`.

bugtraq.provideruuid

This is the GUID of 32-bit issue tracker plugin.

bugtraq.provideruuid64

This is the GUID of 64-bit issue tracker plugin.

bugtraq.providerparams

This is the parameter string for the issue tracker plugin.

This issue tracker integration is not restricted to TortoiseGit; it can be used with other clients (e.g. TortoiseSVN). For more information, read the full *Issue Tracker Integration Specification* [https://gitlab.com/tortoisegit/tortoisegit/blob/master/doc/issuetrackers.txt] in the TortoiseGit source repository. (Section 3, “TortoiseGit is free!” explains how to access the repository).

2.35.2. Getting Information from the Issue Tracker

The previous section deals with adding issue information to the log messages. But what if you need to get information from the issue tracker? The commit dialog has a Windows COM interface which allows integration an external program that can talk to your tracker. Typically you might want to query the tracker to get a list of open issues assigned to you, so that you can pick the issues that are being addressed in this commit.

Any such interface is of course highly specific to your system, so we cannot provide this part, and describing how to create such a program is beyond the scope of this manual. The interface definition and sample programs can be obtained from the `contrib` folder in the *TortoiseGit repository* [https://gitlab.com/tortoisegit/tortoisegit/tree/master/contrib/issue-tracker-plugins]. (Section 3, “TortoiseGit is free!” explains how to access the repository). A summary of the API is also given in Appendix B, *IBugTraqProvider interface*.

For illustration purposes, let's suppose that your system administrator has provided you with an issue tracker plugin which you have installed, and that you have set up some of your working trees to use the plugin in TortoiseGit's settings dialog. When you open the commit dialog from a working tree to which the plugin has been assigned, you will see a new button at the top of the dialog.

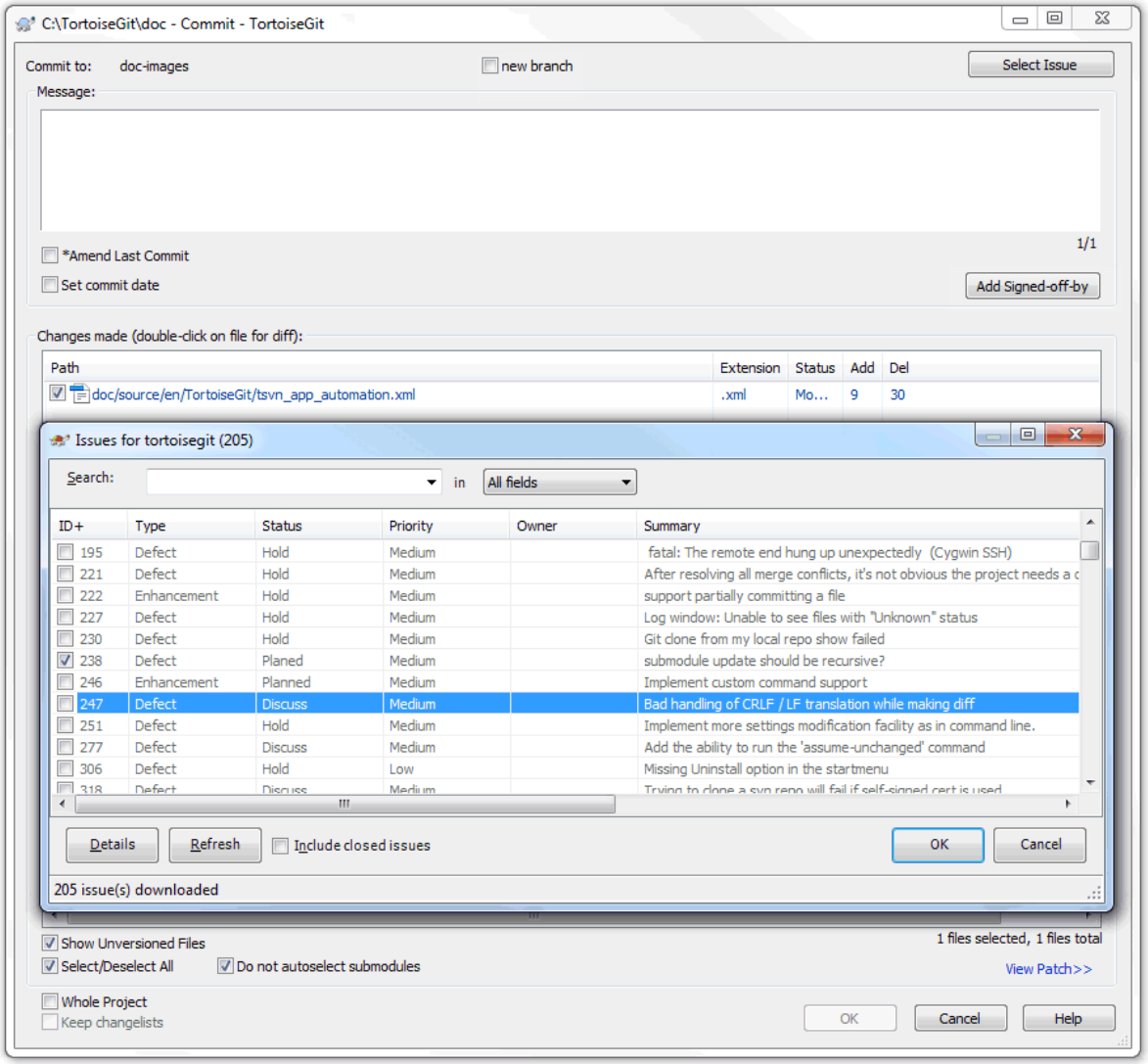


Figure 2.72. Example issue tracker query dialog

In this example you can select one or more open issues. The plugin can then generate specially formatted text which it adds to your log message.

2.36. TortoiseGit's Settings

To find out what the different settings are for, just leave your mouse pointer a second on the textbox/checkbox... and a helpful tooltip will popup.

2.36.1. General Settings

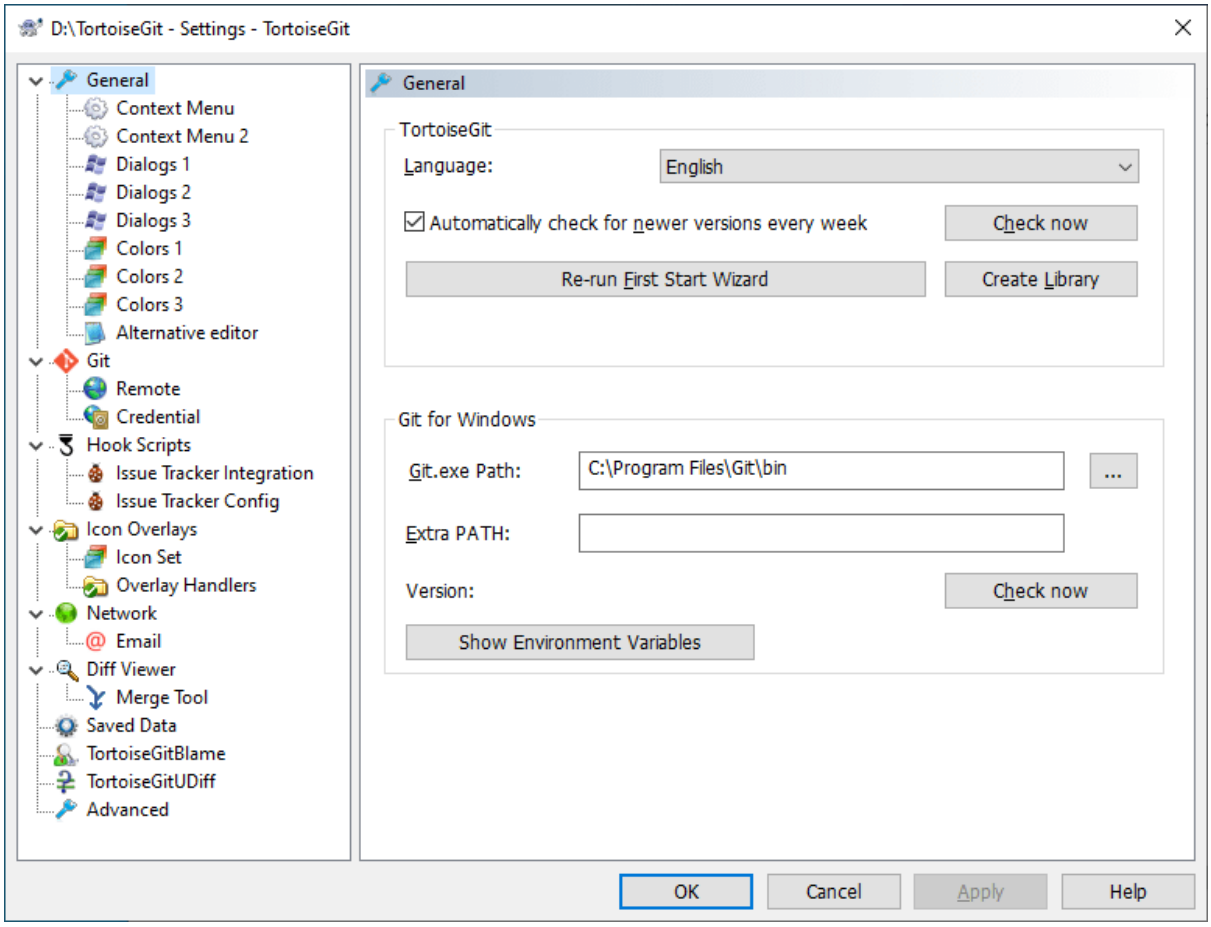


Figure 2.73. The Settings Dialog, General Page

This dialog allows you to specify your preferred language, and the Git-specific settings.

Language

Selects your user interface language. What else did you expect? Only languages of installed language packs are listed. You can download language packs on the TortoiseGit download page [https://tortoisegit.org/download] or help translating [https://tortoisegit.org/translate].

Automatically check for newer versions every week

If checked, TortoiseGit will contact its download site once a week to see if there is a newer version of the program available. Use Check now if you want an answer right away. The new version will not be downloaded; you simply receive an information dialog telling you that the new version is available.

Create Library

On Windows 7 you can create a Library in which to group working copies which are scattered in various places on your system.

Git.exe Path

TortoiseGit needs to know which `git.exe` to use for it's operations. Enter the full path to `git.exe` here.

Caution

`git.exe` must not be marked to be run in elevated mode (i.e. Run as administrator or run in any compatibility mode).

Caution

There is a *known issue in msysGit/Git for Windows* [https://github.com/msysgit/msysgit/issues/103]: Git for Windows provides two `git.exe`-files (one in a folder named `bin` and one in a folder named `cmd`). Make sure Git.exe Path points to the `bin`-folder within the Git for Windows installation folder.

Caution

If you don't use Git for Windows, please see the sections for "Cygwin Git" and "MSYS2 Git" below as special settings are required here.

As a general note: There is no official support [https://tortoisegit.org/support/faq/#nongitforwindowsunsupported] for Cygwin or MSYS2 Git in TortoiseGit. The TortoiseGit developers only use Git for Windows. Bug reports, however, are welcome.

Tip

In order to debug problems you can open TortoiseGit advanced settings and set `DebugOutputString` to "true" (Section 2.36.10, “Advanced Settings”). Start capturing the debug output. Then start TortoiseGit settings, click on Check now and observe the debug messages.

Extra PATH

If your git installation needs an extra entry in the PATH environment variable, you can enter it here and it will get added to the PATH environment variable automatically when TortoiseGit starts.

This is especially needed if you installed the developer version of msysGit ("Full installer (self-contained) if you want to hack on Git" with the filename `msysGit-fullinstall-*.exe`), in this case it is necessary that the `[MSYSGIT-INSTALL-PATH]\mingw\bin`-folder is on the path (i.e. entered in the Extra PATH textbox) in order to execute `git.exe`.

Often you can see if you need this when you start `git.exe` in `[MSYSGIT-INSTALL-PATH]\mingw\bin`-folder and you get a message box saying that a DLL is missing.

Cygwin Git

As noted above: There is no official support [https://tortoisegit.org/support/faq/#nongitforwindowsunsupported] for Cygwin Git in TortoiseGit (do not enable this for the "Git for Windows" package!). The TortoiseGit developers only use Git for Windows. Bug reports, however, are welcome. If you really want to use it, here are the steps you have to perform:

1) Select the `[CYGWIN-INSTALL-PATH]\bin`-folder as `git.exe` folder.

2) Configure the `HOME` environment variable in Windows, so that Cygwin and TortoiseGit are using the same home directory and global git-config. Use the normal Windows notation here (e.g., "C:\Users\USERNAME"). By default, TortoiseGit uses the Windows home directory which is normally located under `c:\Users` and Cygwin uses its own home directories which are located under `[CYGWIN-INSTALL-PATH]\home`.

3) Configure AutoCrLf, this is necessary as TortoiseGit and Cygwin Git have different defaults. The default in Cygwin Git is `true`.

4) Go to TortoiseGit Section 2.36.10, “Advanced Settings” and set `CygwinHack` to `true` in order to activate Cygwin workarounds.

5) Reboot.

MSYS2 Git

As noted above: There is no official support [https://tortoisegit.org/support/faq/#nongitforwindowsunsupported] for MSYS2 Git in TortoiseGit (do not enable this for the "Git for Windows" package!). The TortoiseGit developers only use Git for Windows. Bug reports, however, are welcome. If you really want to use it, here are the steps you have to perform:

1) Select the `[MSYS2-INSTALL-PATH]\usr\bin`-folder as `git.exe` folder.

2) Configure the `HOME` environment variable in Windows, so that MSYS2 and TortoiseGit are using the same home directory and global git-config. Use the normal Windows notation here (e.g., `C:\Users\USERNAME`). By default, TortoiseGit uses the Windows home directory which is normally located under `c:\Users` and MSYS2 uses its own home directories which are located under `[MSYS2-INSTALL-PATH]\home`.

3) Configure AutoCrLf, this is necessary as TortoiseGit and MSYS2 Git might have different defaults.

4) Go to TortoiseGit Section 2.36.10, “Advanced Settings” and set `Msys2Hack` to `true` in order to activate MSYS2 workarounds.

5) Reboot.

2.36.1.1. Context Menu Settings

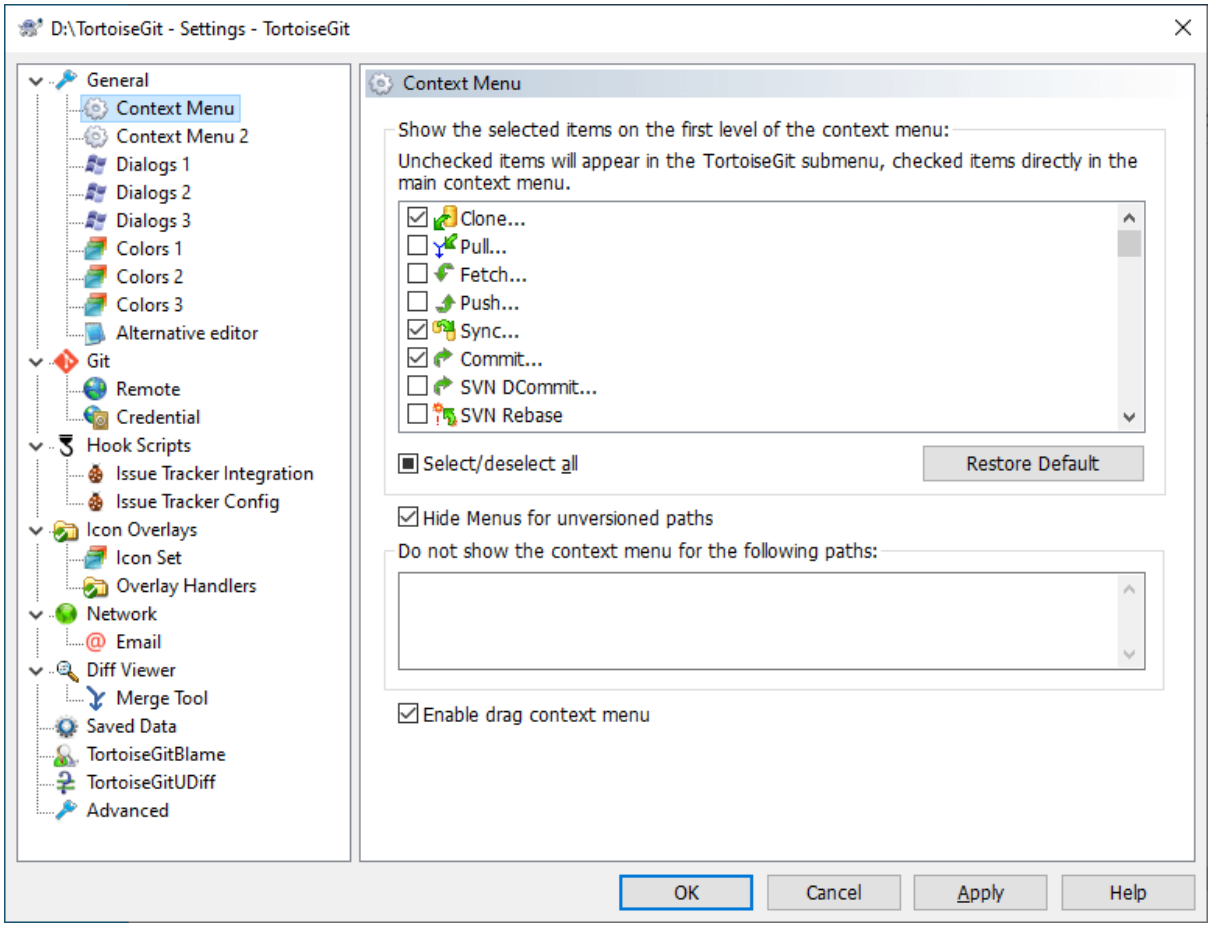


Figure 2.74. The Settings Dialog, Context Menu Page

This page allows you to specify which of the TortoiseGit context menu entries will show up in the main context menu (on the first level), and which entries will appear in the TortoiseGit submenu. By default most items are

unchecked and appear in the submenu. If you want to hide specific entries, see Section 2.36.1.2, “Context Menu 2 Settings”.

Most of the time, you won't need the TortoiseGit context menu, apart for folders that are under version control by Git. For non- versioned folders, you only really need the context menu when you want to do a checkout. If you check the option Hide menus for unversioned paths, TortoiseGit will not add its entries to the context menu for unversioned folders. But the entries are added for all items and paths in a versioned folder. And you can get the entries back for unversioned folders by holding the **Shift** key down while showing the context menu.

If there are some paths on your computer where you just don't want TortoiseGit's context menu to appear at all, you can list them in the box at the bottom.

If you right click and drag folder/file in Windows Explorer, a context menu will be shown when you drop. It provides some TortoiseGit actions. You can uncheck Enable drag context menu to prevent from carelessly clicking the TortoiseGit actions.

There's a different settings page for the Windows 11 context menu. That page also has a button to register the TortoiseGit entries in the context menu. You only have to do this if TortoiseGit was installed as a different user.

2.36.1.2. Context Menu 2 Settings

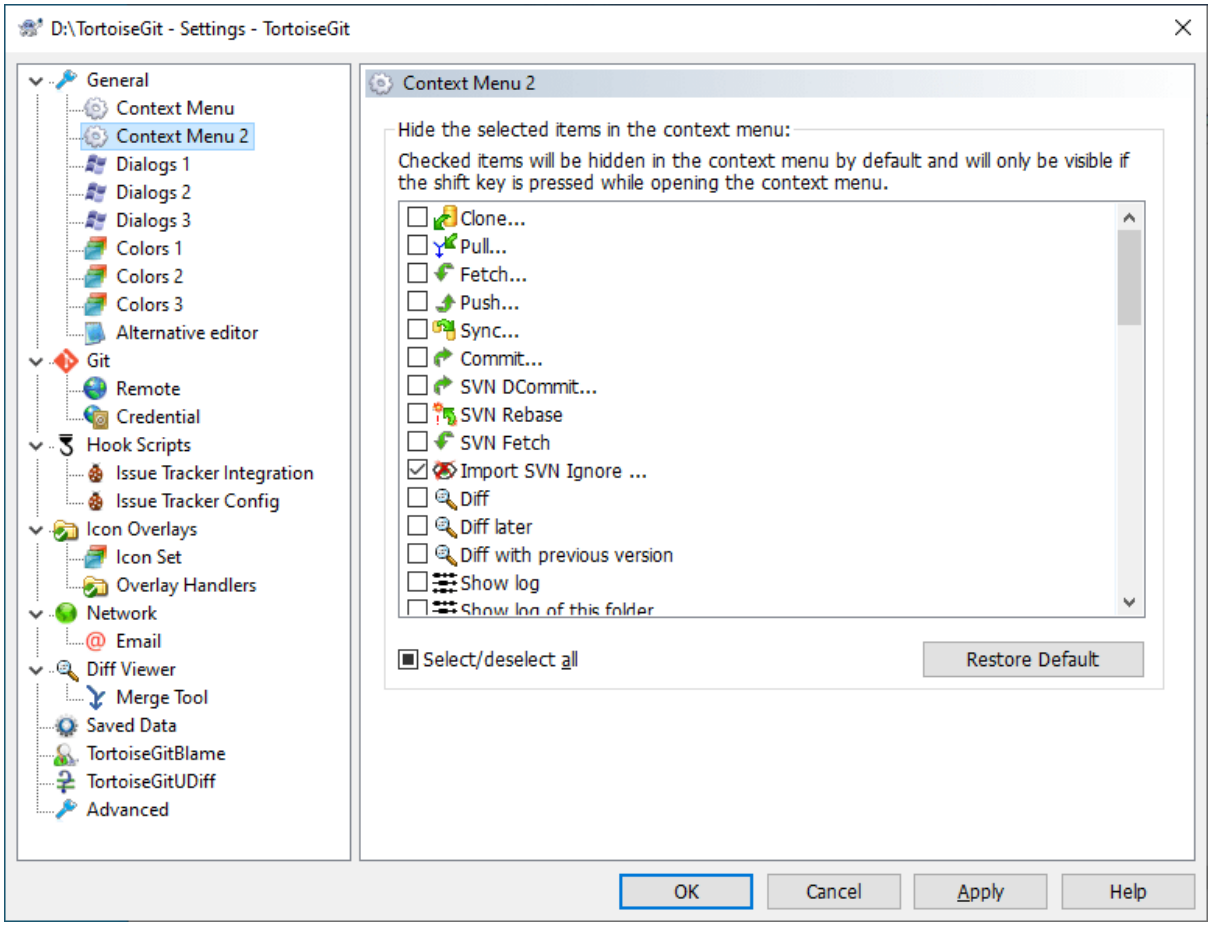


Figure 2.75. The Settings Dialog, Context Menu 2

This page allows you to specify which of the TortoiseGit context menu entries will be hidden by default. Selected item will only be visible when you hold the **Shift** key on right click (this is the so-called extended context menu, please don't mix this with the TortoiseGit submenu, which is also configurable (cf. Section 2.36.1.1, “Context Menu Settings”)). This configuration helps you to reduce the number of context menu entries according to your needs.

2.36.1.3. TortoiseGit Dialog Settings

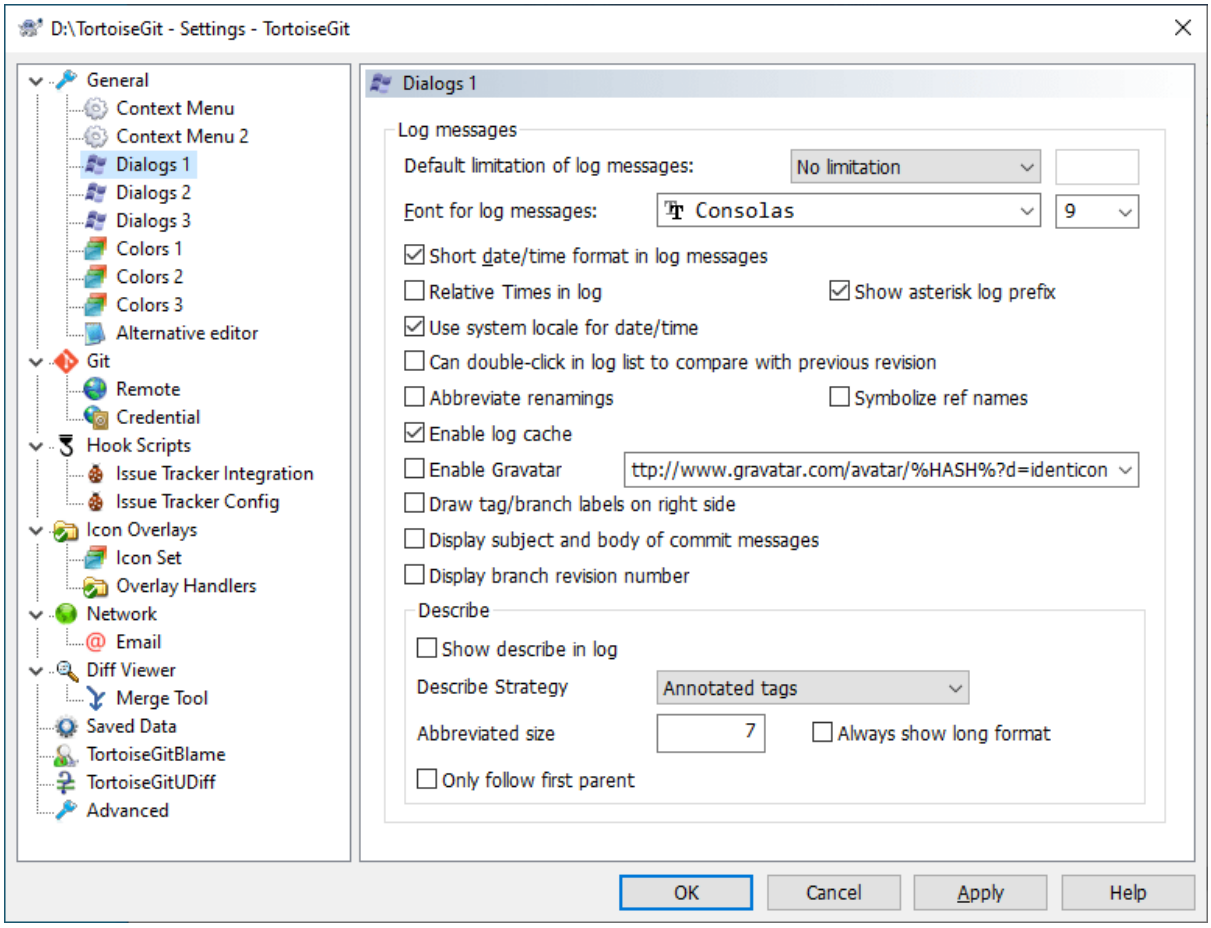


Figure 2.76. The Settings Dialog, Dialogs Page

This dialog allows you to configure some of TortoiseGit's dialogs the way you like them.

Default limitation of log messages

Limits the number of log messages that TortoiseGit loads and displays when you first select TortoiseGit → Show log... (cf. Section 2.14, "Log Dialog"). The number of log messages can be limited by a fixed number of revisions to display, it can remember the last selected date when the Log Dialog is opened again, it can be limited to the last N year(s)/month(s)/week(s), or just be unlimited (i.e., all revisions are shown; this is the default). When a default limitation is active, the From label has a blueish background in the Revision Log dialog. By opening the context menu on the From date picker, you can configure the default limit or disable the default limit for the current dialog.

Font for log messages

Selects the font face and size used to display the log message itself in the middle pane of the Revision Log dialog, and when composing log messages in the Commit dialog.

Short date / time format in log messages

If the standard long messages use up too much space on your screen use the short format.

Show asterisk log prefix

An asterisk is inserted as the prefix of log message in Log dialog.

Can double-click in log list to compare with previous revision

If you frequently find yourself comparing revisions in the top pane of the log dialog, you can use this option to allow that action on double-click. It is not enabled by default because fetching the diff is often a long process, and many people prefer to avoid the wait after an accidental double-click, which is why this option is not enabled by default.

Abbreviate renamings

Normally renamed files are listed as `long/path/for/file.txt (from long/path/to/file.txt)`. If you check this option renamed files will be listed in a shorter format (`long/path/{to => for}/file.txt`), however, this abbreviated format might be harder to understand.

Symbolize ref names

Show symbols on ref labels to substitute part of the ref names in order to make them smaller. If this option is enabled, the following description and example will apply. If there is only a single remote, an up-arrow symbol (↑) will substitute the remote name part of each remote branch. If the remote branch is the upstream of a local branch, an equivalent symbol (≡) will substitute the branch name part of the remote branch.

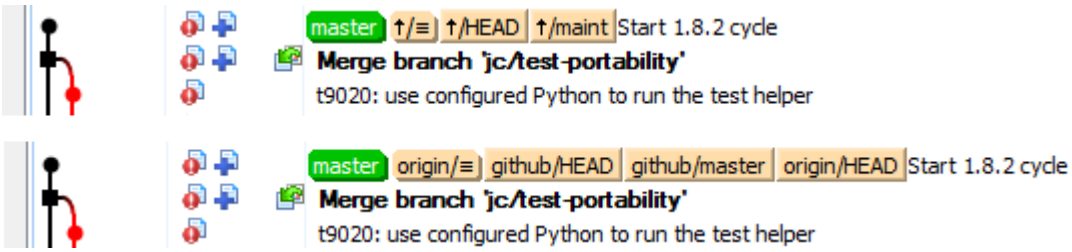


Figure 2.77. Example of Symbolize ref names

Enable log cache

Load/saves log cache in `.git` folder (`tortoisegit.data`, `tortoisegit.index`) to boost performance of subsequent use of log list. If this option is disabled, the cache files are not read or written. Default is enabled.

Enable Gravatar

Shows the Gravatar image of the author of the commit in Log Dialog. The URL is customizable so you may specify more options supported by the server, or use your own avatar server. The default URL is `https://www.gravatar.com/avatar/%HASH%?d=identicon` Currently, the supported parameter is `%HASH%`, which is the MD5 email hash. To specify a default image, add `d=` parameter, e.g. `https://www.gravatar.com/avatar/%HASH%?d=identicon` See *Gravatar: Image Requests* [https://en.gravatar.com/site/implement/images/] for a list of parameters.

Draw tag/branch labels on right side

Shows tag/branch labels after the commit message.

Display branch revision number

Displays for every selected commit a so called "branch revision number" in the commit message field of the Log Dialog. The branch revision number is calculated by calling `git rev-list --count --first-parent [SHA1]` and represents the number of commits between the beginning of time and the selected commit. This number is NOT guaranteed to be unique, especially if you alter the history (e.g., using rebase) or use several branches at the same time. It can be seen "kinda unique" per branch in case you don't alter its history (e.g. by rebasing, resetting) and only commit or merge other branches on it. This number is only displayed for first-parent commits and not for commits on non-fast-forward merges (here duplicate numbers could occur). See *https://gcc.gnu.org/ml/gcc/2015-08/msg00148.html* [https://gcc.gnu.org/ml/gcc/2015-08/

msg00148.html] and *https://gitlab.com/tortoisegit/tortoisegit/merge_requests/1* [https://gitlab.com/tortoisegit/tortoisegit/merge_requests/1] for more details.

Show describe in log

Shows describe above commit message in the Log dialog. For example, `v0.21.0-589-gdeadc43` refers to the commit `deadc43` that is 589 commits ahead the tag `v0.21.0`. Note: Describe may take longer to run if the commit is far ahead away from a tag.

Describe strategy

Determine reference lookup strategy: Available options: Annotated tags, All tags, All refs. Default strategy is annotated tags only. If your repository uses lightweight tags to mark releases, choose All tags.

Describe Abbreviated size

Number of chars of the abbreviated commit id to show in describe. Default is 7.

Describe Always show long format

Whether to use the long format even when a shorter name could be used. For example, when the commit `g28f087c` has tag `v0.21.0`, it still shows long format `v0.21.0-0-g28f087c` instead of just `v0.21.0`.

2.36.1.4. TortoiseGit Dialog Settings 2

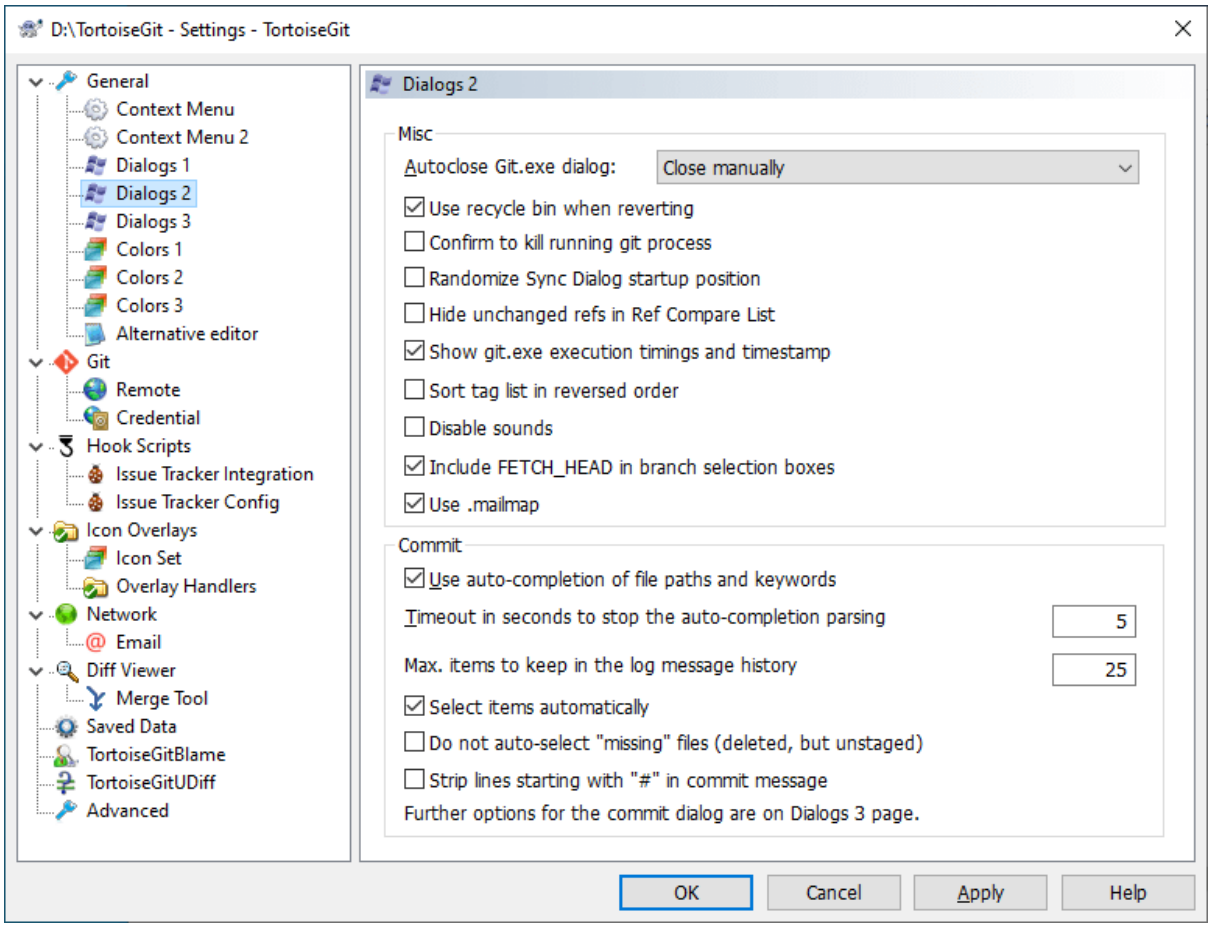


Figure 2.78. The Settings Dialog, Dialogs Page 2

This dialog allows you to configure some more of TortoiseGit's dialogs the way you like them.

Git.exe Progress Dialog

TortoiseGit can automatically close all progress dialogs when the action is finished without error. This setting allows you to select the conditions for closing the dialogs. The default (recommended) setting is Close manually which allows you to review all messages and check what has happened. However, you may decide that you want to ignore some types of message and have the dialog close automatically if there are no critical changes.

Auto-close if no further options are available will close the dialog if `git.exe` exited cleanly (i.e. no error occurred) and no further options are presented in the progress dialog.

Auto-close if no errors always closes the dialog if `git.exe` exited with 0 error code.

Use recycle bin when reverting

When you revert local modifications, your changes are discarded. TortoiseGit gives you an extra safety net by sending the modified file to the recycle bin before bringing back the pristine copy. If you prefer to skip the recycle bin, uncheck this option.

Confirm to kill running git process

When enabled, if you close Progress Dialog or Sync Dialog with a running git process, you will be asked for confirmation before killing it. This avoids closing the dialog by accident that kills running git process.

Randomize Sync Dialog startup position

When enabled, the startup position of Sync Dialog will be randomized. If you open many Sync Dialogs and press pull button at the same time, you may easily press the pull button in any previous Sync Dialog if it finishes and becomes foreground.

Hide unchanged refs in Ref Compare List

When enabled, unchanged refs will not be shown in Ref Compare List, so you can focus on changed refs. Currently, this list is in Sync Dialog Ref List tab.

Show `git.exe` execution timings and timestamp

When enabled, `git.exe` execution timings and timestamp will be appended at the end of progress message.

Sort tag list in reversed order

When enabled, tag list is sorted in reversed order. It is because newer versions are more useful. e.g. Export Dialog allows to select the latest tag when this option is enabled.

Use auto-completion of file paths and keywords

The commit dialog includes a facility to parse the list of filenames being committed. When you type the first 3 letters of an item in the list, the auto-completion box pops up, and you can press Enter to complete the filename. Check the box to enable this feature.

Timeout in seconds to stop the auto-completion parsing

The auto-completion parser can be quite slow if there are a lot of large files to check. This timeout stops the commit dialog being held up for too long. If you are missing important auto-completion information, you can extend the timeout.

Max. items to keep in the log message history

When you type in a log message in the commit dialog, TortoiseGit stores it for possible re-use later. By default it will keep the last 25 log messages for each repository, but you can customize that number here. If you have many different repositories, you may wish to reduce this to avoid filling your registry.

Note that this setting applies only to messages that you type in on this computer. It has nothing to do with the log cache.

Select items automatically

The normal behavior in the commit dialog is for all modified (versioned) items to be selected for commit automatically. If you prefer to start with nothing selected and pick the items for commit manually, uncheck this box.

2.36.1.5. TortoiseGit Dialog Settings 3

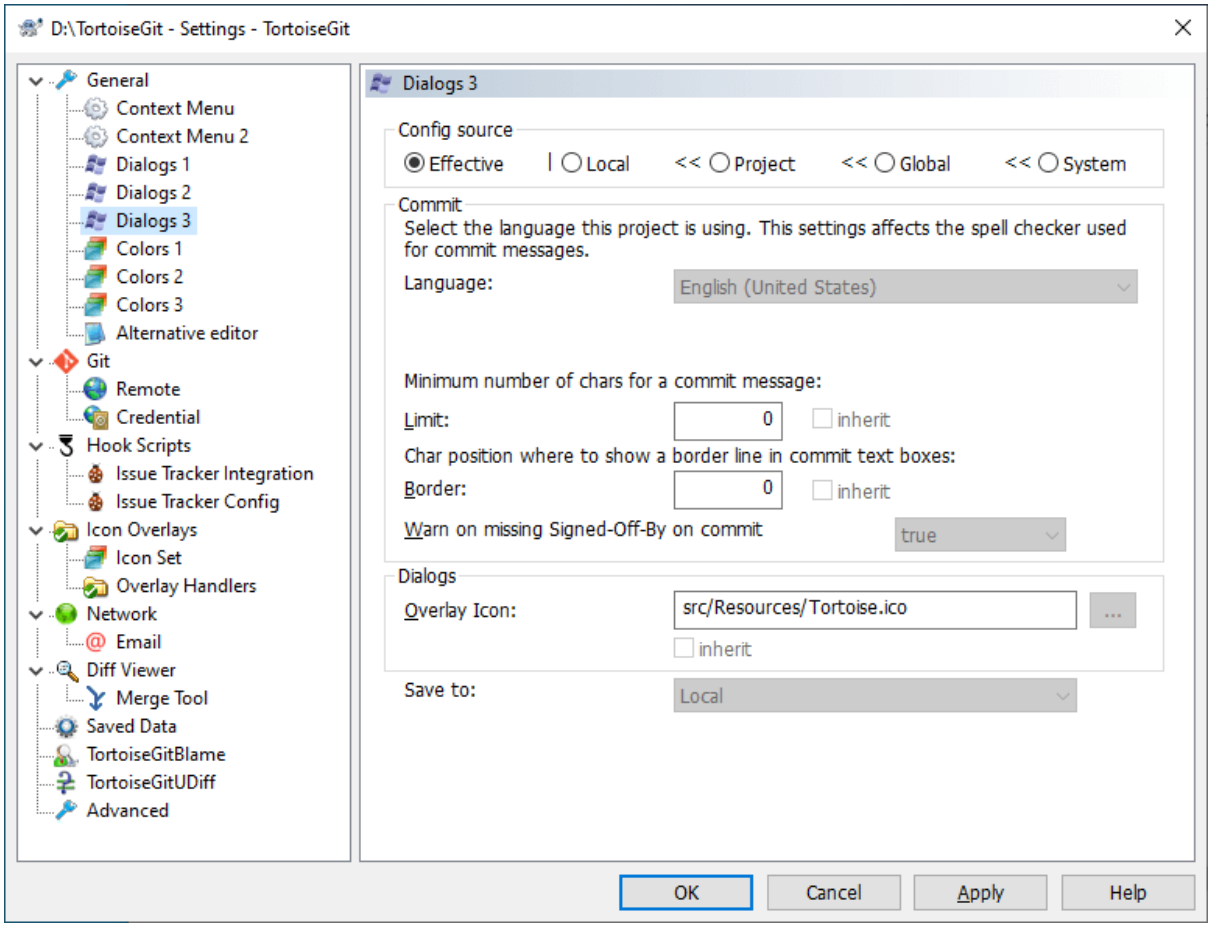


Figure 2.79. The Settings Dialog, Dialogs 3 Page

This dialog allows you to configure some of TortoiseGit's dialogs the way you like them. This third page mainly affects the Commit dialog and the settings which are stored in git config files.

> **Important**
>
> If you have problems entering/storing data please see Section 2.36.6.1, “The hierarchical Git configuration”.

Language

TortoiseGit by default uses the spell checker modules which are also used by OpenOffice, LibreOffice and Mozilla. Optionally, the Windows 8+ spell checker can also be used (needs to be enabled manually at the moment). If you have those installed or use the Windows spell checker this property will determine which spell

checker to use, i.e. in which language the log messages for your project should be written. The `tgit.projectlanguage` config key sets the language module the spell checking engine should use when you enter a log message. You can find the values for your language on this page: *MSDN: Language Identifiers* [https://docs.microsoft.com/windows/desktop/intl/language-identifier-constants-and-strings].

Enter this value in decimal. For example English (US) can be entered as `1033`.

Use `-1` to disable the spell checker.

Limit

`tgit.logminsize` sets the minimum length of a log message for a commit. If you enter a shorter message than specified here, the commit button is disabled. This feature is very useful for reminding you to supply a proper descriptive message for every commit. If this property is not set, or the value is zero, empty log messages are allowed.

Border

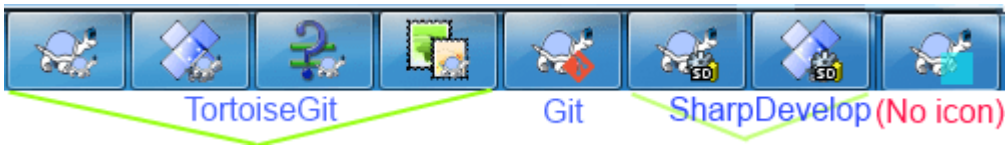
`tgit.logwidthmarker` is used with projects which require log messages to be formatted with some maximum width (typically 72 characters) before a line break. Setting this property to a non-zero will place a marker to indicate the maximum width and performs line wrapping. Note: this feature will only work correctly if you have a fixed-width font selected for log messages.

Warn on Signed-Off-By on commit

`tgit.warnnosignedoffby` is used with projects which require Signed-off-by line in commit messages.

Overlay Icon

`tgit.icon` is used with projects which wish to show the logo on the taskbar for easier identification when multiple TortoiseGit application instances of different projects are running at the same time.



If icon is not 16x16 pixels in size, it will be automatically scaled. Supported formats are `.ico`, `.png`, `.jpg`, `.gif`, `.bmp`. If no icon is included by that project, you may find one on you own, put it in `.git` folder and set the relative path in local config. e.g. `.git/logo.ico` If you want to disable it, you may set `tgit.icon` as an empty string in local config. It will fallback to a color block when disabled or load failed. Note that the advanced option `GroupTaskbarIconsPerRepo` should be 3 or 4 in order to use this function.

2.36.1.6. TortoiseGit color Settings

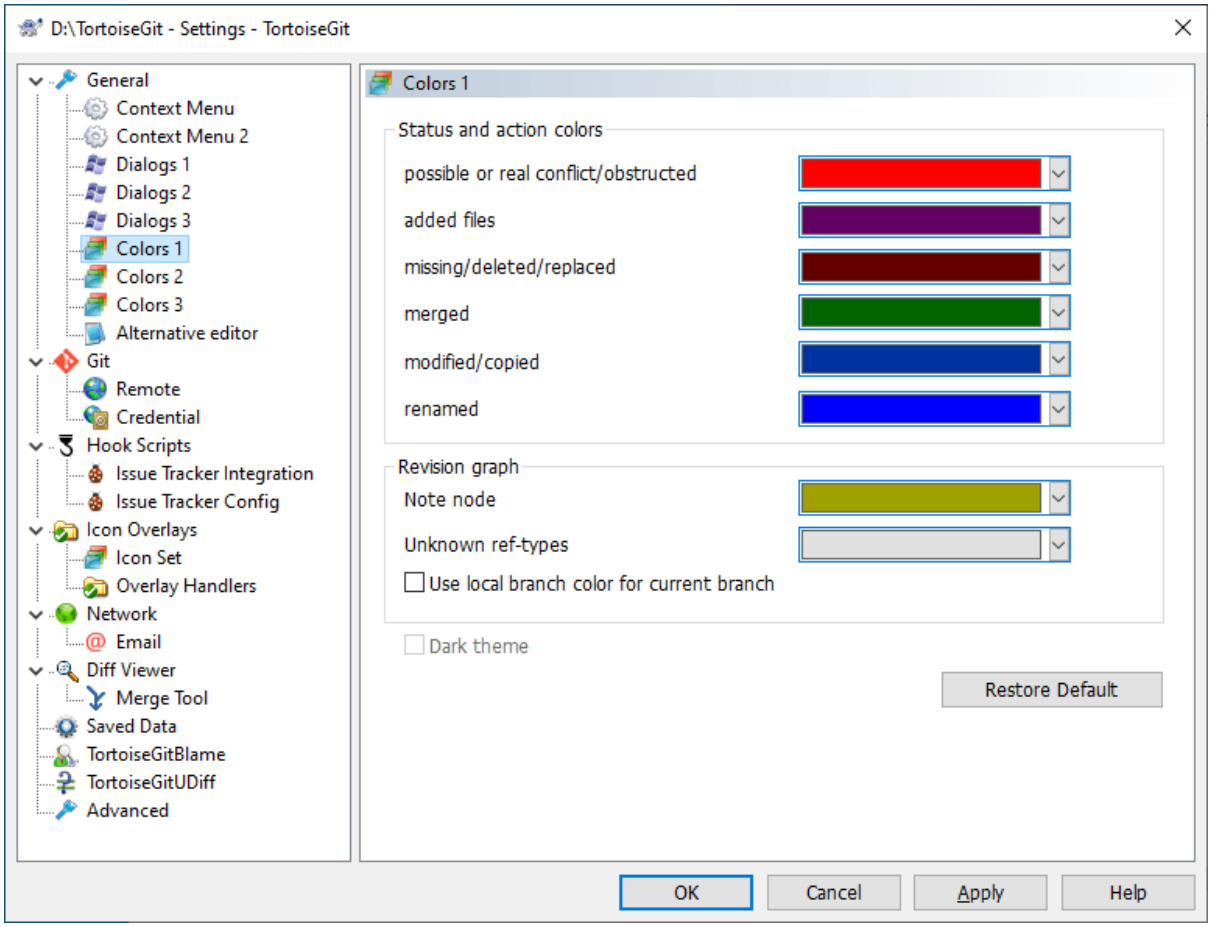


Figure 2.80. The Settings Dialog, colors Page

This dialog allows you to configure the text colors used in TortoiseGit's dialogs the way you like them.

Possible or real conflict / obstructed

A conflict has occurred during update, or may occur during merge. Update is obstructed by an existing unversioned file/folder of the same name as a versioned one.

This color is also used for error messages in the progress dialogs.

Added files

Items added to the repository.

Missing / deleted / replaced

Items deleted from the repository, missing from the working copy, or deleted from the working tree and replaced with another file of the same name.

Merged

Changes from the repository successfully merged into the working tree without creating any conflicts.

Modified / copied

Add with history, or paths copied in the repository. Also used in the log dialog for entries which include copied items.

Note node

A reference which points to git notes, under refs/notes name space.

Use local branch color for current branch

In revision graph, use local branch color for current branch. You may not want to emphasize current branch of a local repository in revision graph.

other settings:

Dark theme

The dialogs in TortoiseGit can be shown in a dark mode on Windows 10 1809 and later. This feature also requires that dark mode for applications is enabled in the Windows 10 settings.

Important

Note that not all controls in all dialogs are shown in a dark theme.

2.36.1.7. TortoiseGit color Settings 2

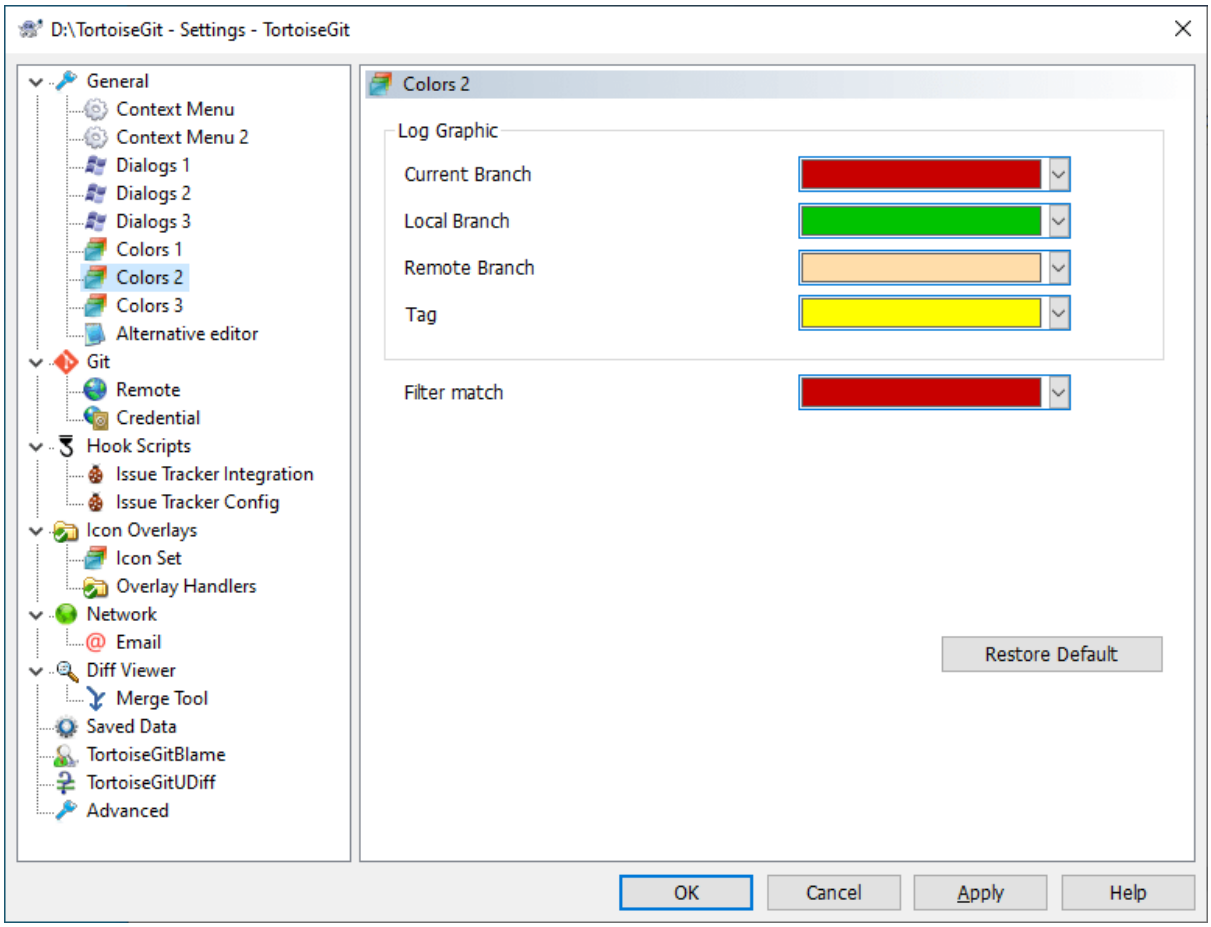


Figure 2.81. The Settings Dialog, colors Page

This dialog allows you to configure the text colors used in TortoiseGit's dialogs the way you like them.

2.36.1.8. TortoiseGit color Settings 3

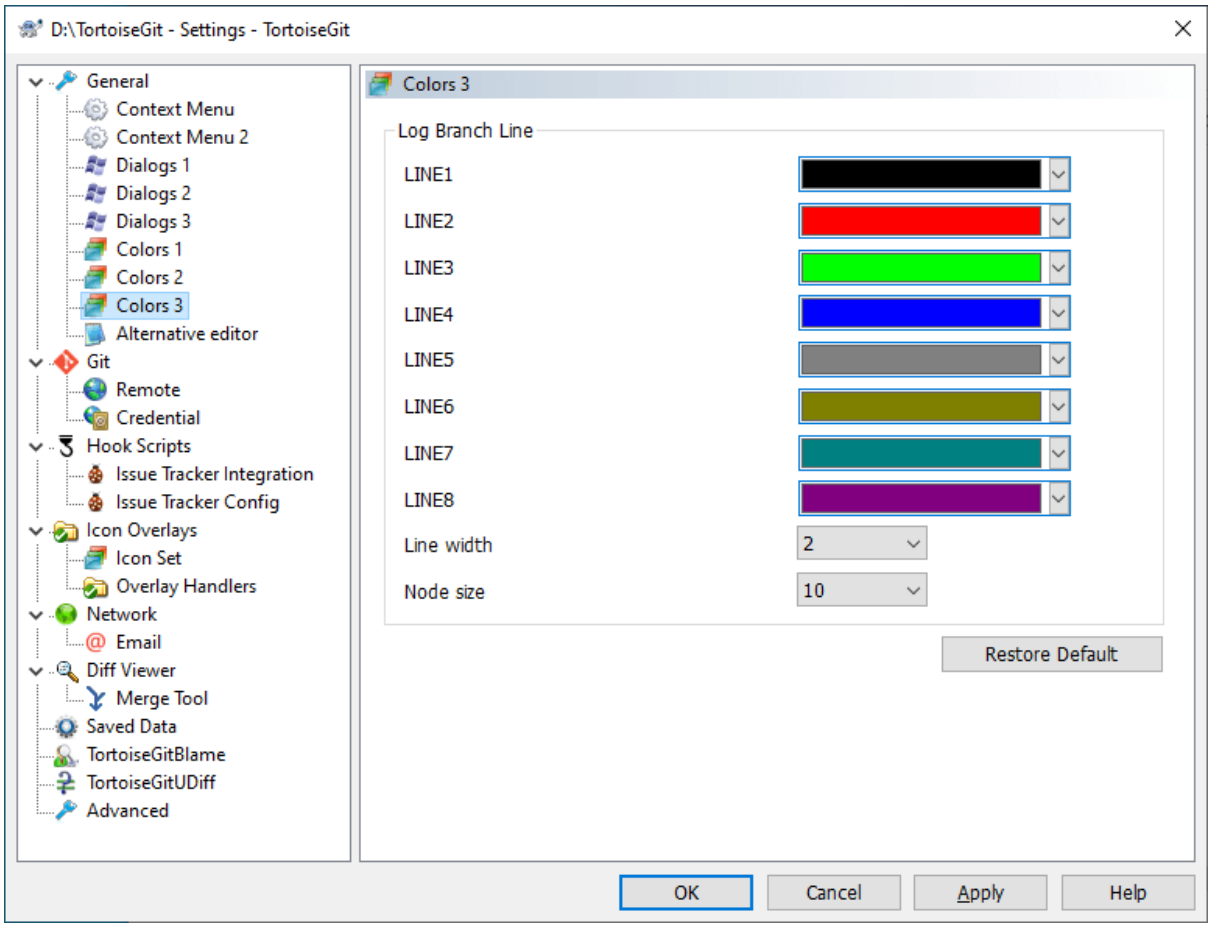


Figure 2.82. The Settings Dialog, colors Page

This dialog allows you to configure the line colors, line width and node size in the graph column used in TortoiseGit's log dialog the way you like them.

2.36.2. Icon Overlay Settings

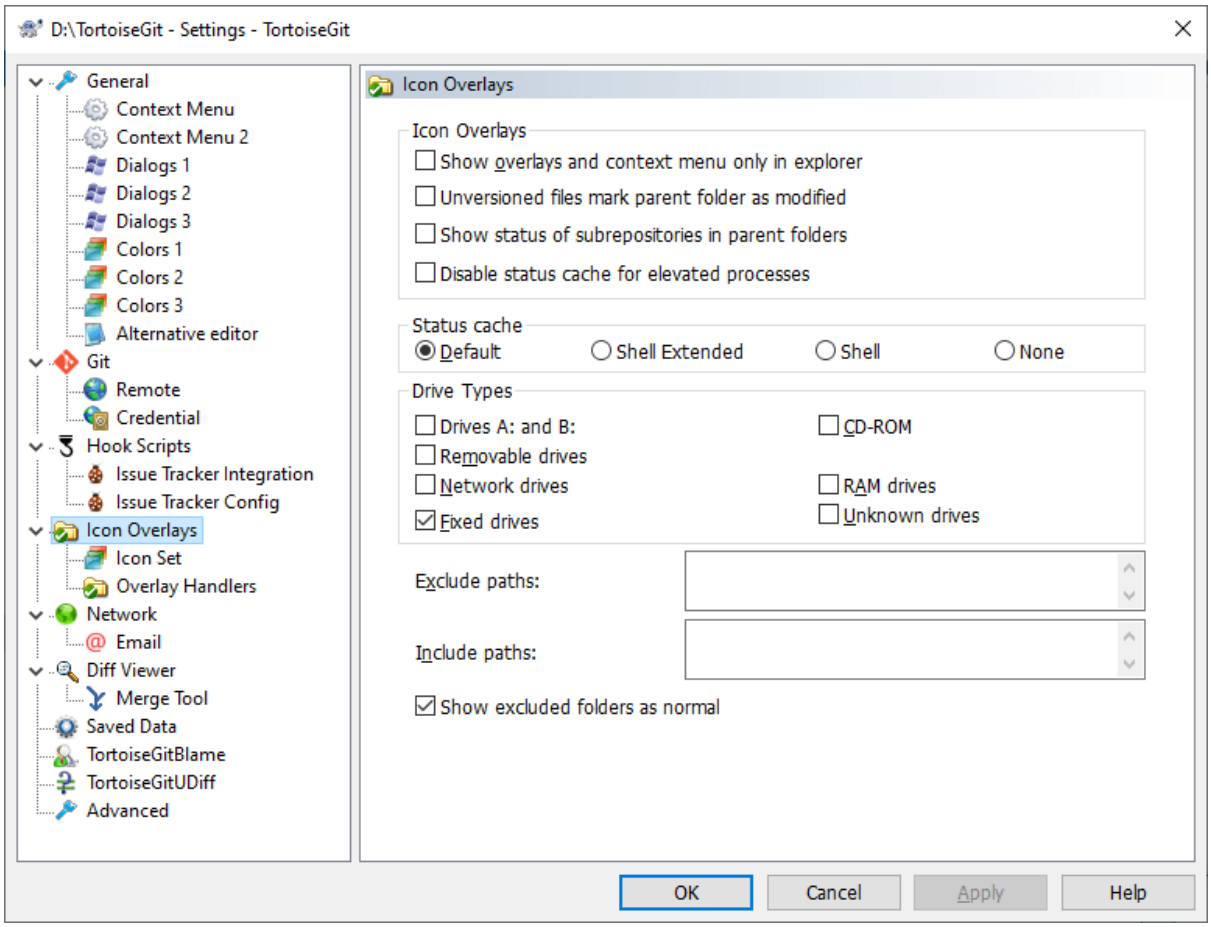


Figure 2.83. The Settings Dialog, Icon Overlays Page

This page allows you to choose the items for which TortoiseGit will display icon overlays.

By default, overlay icons and context menus will appear in all open/save dialogs as well as in Windows Explorer. If you want them to appear *only* in Windows Explorer, check the Show overlays and context menu only in explorer box.

Ignored items and Unversioned items are not usually given an overlay. If you want to show an overlay in these cases, just check the boxes.

You can also choose to mark folders as modified if they contain unversioned items. This could be useful for reminding you that you have created new files which are not yet versioned. This option is only available when you use the *default* status cache option (see below).

Since it takes quite a while to fetch the status of a working tree, TortoiseGit uses a cache to store the status so the explorer doesn't get hogged too much when showing the overlays. You can choose which type of cache TortoiseGit should use according to your system and working tree size here:

Default

Caches all status information in a separate process (`TGitCache.exe`). That process watches all drives for changes and fetches the status again if files inside a working tree get modified. The process runs with the least possible priority so other programs don't get hogged because of it. That also means that the status information is not *real time* but it can take a few seconds for the overlays to change.

Advantage: the overlays show the status recursively, i.e. if a file deep inside a working tree is modified, all folders up to the working tree root will also show the modified overlay. And since the process can send notifications to the shell, the overlays on the left tree view usually change too.

Disadvantage: the process runs constantly, even if you're not working on your projects. It also uses around 10-50 MB of RAM depending on number and size of your working trees. From version 1.7.0 to 1.7.12 TGitCache did not check the contents of the files, it just checked the last modification time against the time stored in the git index file. Starting from 1.7.13 TGitCache now also checks the contents of the files by default. If you want to restore the old behavior, you can disable checking the contents via the Settings dialog -> Advanced and set `TGitCacheCheckContentMaxSize` to "0".

Shell Extended

Caching is done directly inside the shell extension DLL. Each time you navigate to another folder, the status information is fetched again (recursively).

Advantage: can show the status in *real time*.

Disadvantage: only one folder is cached and for big working trees, it can take much more time to show a folder in explorer than with the default cache or with shell mode. The Shell variant only shows differences of the filesystem to the git index (does not include revision specific information, e.g. if you remove a file from the index the file will show up as unversioned, but with TGitCache the file will show up as deleted until you commit this change).

Shell

Caching is done directly inside the shell extension DLL, but only for the currently visible folder. Each time you navigate to another folder, the status information is fetched again.

Advantage: needs only very little memory (around 1 MB of RAM) and can show the status in *real time*.

Disadvantage: Since only one folder is cached, the overlays don't show the status recursively. For big working trees, it can take more time to show a folder in explorer than with the default cache. The Shell variant only shows differences of the filesystem to the git index (does not include revision specific information, e.g. if you remove a file from the index the file will show up as unversioned, but with TGitCache the file will show up as deleted until you commit this change).

None

With this setting, the TortoiseGit does not fetch the status at all in Explorer. Because of that, files don't get an overlay and folders only get a 'normal' overlay if they're versioned. No other overlays are shown, and no extra columns are available either.

Advantage: uses absolutely no additional memory and does not slow down the Explorer at all while browsing.

Disadvantage: Status information of files and folders is not shown in Explorer. To see if your working trees are modified, you have to use the “Check for modifications” dialog.

By default, overlay icons and context menus will appear in all open/save dialogs as well as in Windows Explorer. If you want them to appear *only* in Windows Explorer, check the Show overlays and context menu only in explorer box.

You can force the status cache to *None* for elevated processes by checking the Disable status cache for elevated processes box. This is useful if you want to prevent another `TGitCache.exe` process getting created with elevated privileges.

You can also choose to mark folders as modified if they contain unversioned items. This could be useful for reminding you that you have created new files which are not yet versioned. This option is only available when you use the *default* status cache option (see below).

The next group allows you to select which classes of storage should show overlays. By default, only hard drives are selected. You can even disable all icon overlays, but where's the fun in that?

Network drives can be very slow, so by default icons are not shown for working trees located on network shares.

USB Flash drives appear to be a special case in that the drive type is identified by the device itself. Some appear as fixed drives, and some as removable drives.

The Exclude Paths are used to tell TortoiseGit those paths for which it should *not* show icon overlays and status columns. This is useful if you have some very big working trees containing only libraries which you won't change at all and therefore don't need the overlays, or if you only want TortoiseGit to look in specific folders.

Any path you specify here is assumed to apply recursively, so none of the child folders will show overlays either. If you want to exclude *only* the named folder, append `?` after the path.

The same applies to the Include Paths. Except that for those paths the overlays are shown even if the overlays are disabled for that specific drive type, or by an exclude path specified above.

Users sometimes ask how these three settings interact. For any given path check the include and exclude lists, seeking upwards through the directory structure until a match is found. When the first match is found, obey that include or exclude rule. If there is a conflict, a single directory spec takes precedence over a recursive spec, then inclusion takes precedence over exclusion.

An example will help here:

```
Exclude:
C:
C:\develop\?
C:\develop\tgit\obj
C:\develop\tgit\bin

Include:
C:\develop
```

These settings disable icon overlays for the C: drive, except for `c:\develop`. All projects below that directory will show overlays, except the `c:\develop` folder itself, which is specifically ignored. The high-churn binary folders are also excluded.

TGitCache.exe also uses these paths to restrict its scanning. If you want it to look only in particular folders, disable all drive types and include only the folders you specifically want to be scanned.

Exclude SUBST Drives

It is often convenient to use a SUBST drive to access your working trees, e.g. using the command

```
subst T: C:\TortoiseGit\doc
```

However this can cause the overlays not to update, as `TGitCache` will only receive one notification when a file changes, and that is normally for the original path. This means that your overlays on the `subst` path may never be updated.

An easy way to work around this is to exclude the original path from showing overlays, so that the overlays show up on the `subst` path instead.

Sometimes you will exclude areas that contain working trees, which saves TGitCache from scanning and monitoring for changes, but you still want a visual indication that a folder contains a working tree. The Show excluded folders as 'normal' checkbox allows you to do this. With this option, working tree folders in any excluded area (drive type not checked, or specifically excluded) will show up as normal and up-to-date, with a green check mark. This reminds you that you are looking at a working tree, even though the folder overlays may not be correct. Files do not get an overlay at all. Note that the context menus still work, even though the overlays are not shown.

As a special exception to this, drives `A:` and `B:` are never considered for the Show excluded folders as 'normal' option. This is because Windows is forced to look on the drive, which can result in a delay of several seconds when starting Explorer, even if your PC does have a floppy drive.

2.36.2.1. Icon Set Selection

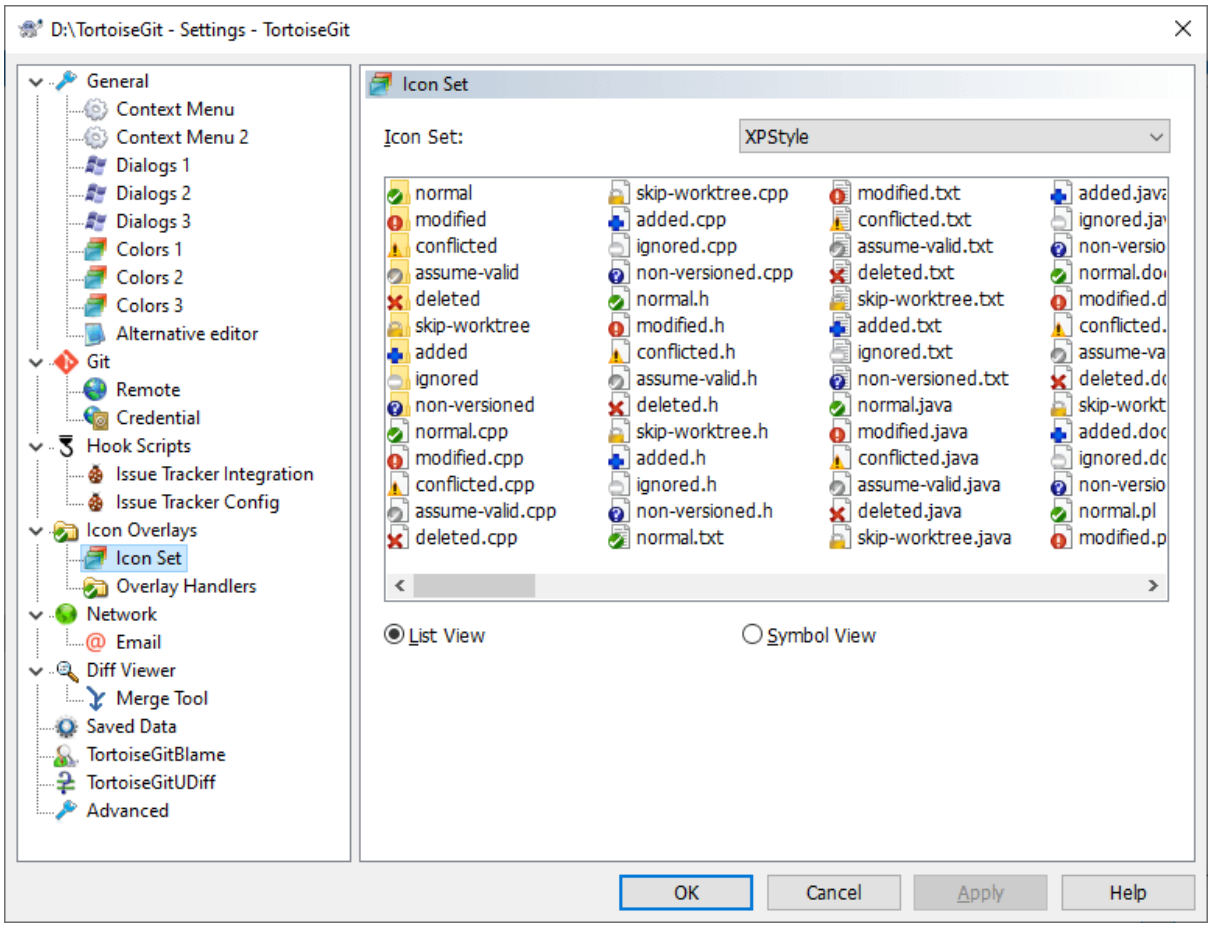


Figure 2.84. The Settings Dialog, Icon Set Page

You can change the overlay icon set to the one you like best. Especially you can disable overlays which you do not need like assume-valid and skip-worktree, however other Tortoise* tools use these two for different purposes. Note that if you change overlay set, you may have to restart your computer for the changes to take effect.

2.36.2.2. Enabled Overlay Handlers

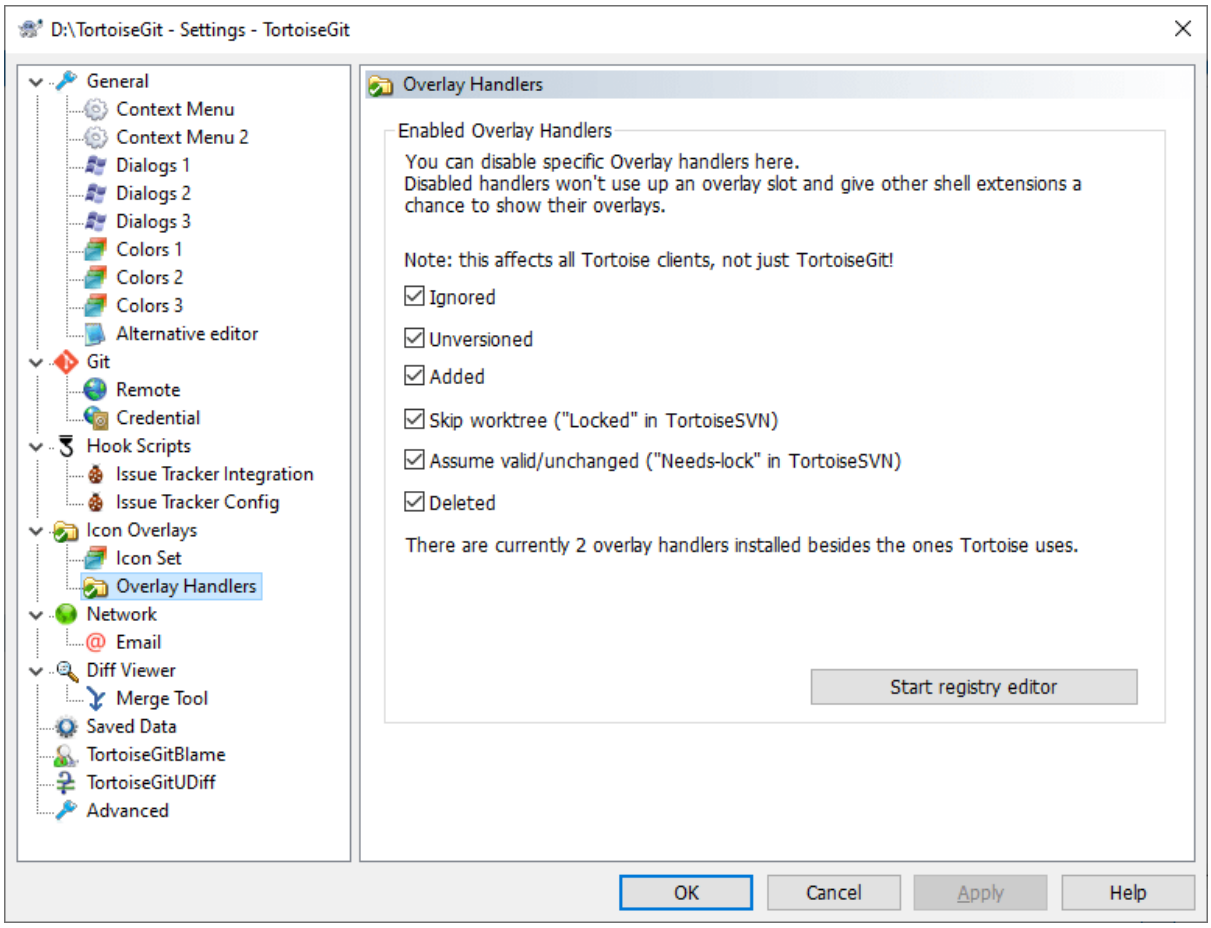


Figure 2.85. The Settings Dialog, Icon Handlers Page

Because the number of overlays available is severely restricted, you can choose to disable some handlers to ensure that the ones you want will be loaded. Because TortoiseGit uses the common TortoiseOverlays component which is shared with other Tortoise clients (e.g. TortoiseSVN, TortoiseCVS, TortoiseHg) this setting will affect those clients too.

For a description of how icon overlays correspond to Git status and other technical details, read Section E.1, “Icon Overlays”.

Windows explorer can just handle a fixed number different overlay providers (15) and TortoiseGit is using 6 of these (these 6 are handled by TortoiseOverlays and, thus, shared with TortoiseSVN and TortoiseCVS). If the TortoiseGit icons are not correctly displayed this is likely caused by other programs which provide overlays (like DropBox, Owncloud, BoxSync and various others) and register with a higher priority. Use the Start registry editor button for opening the registry editor at the key where the overlay handlers are registered. Just delete or rename the ones you don't need OR prepend the `Tortoise` ones with a double quote or space characters so that those come first in the list. For more information please see TortoiseGit FAQ [https://tortoisegit.org/support/faq/#ovlnotshowing].

2.36.3. Network Settings

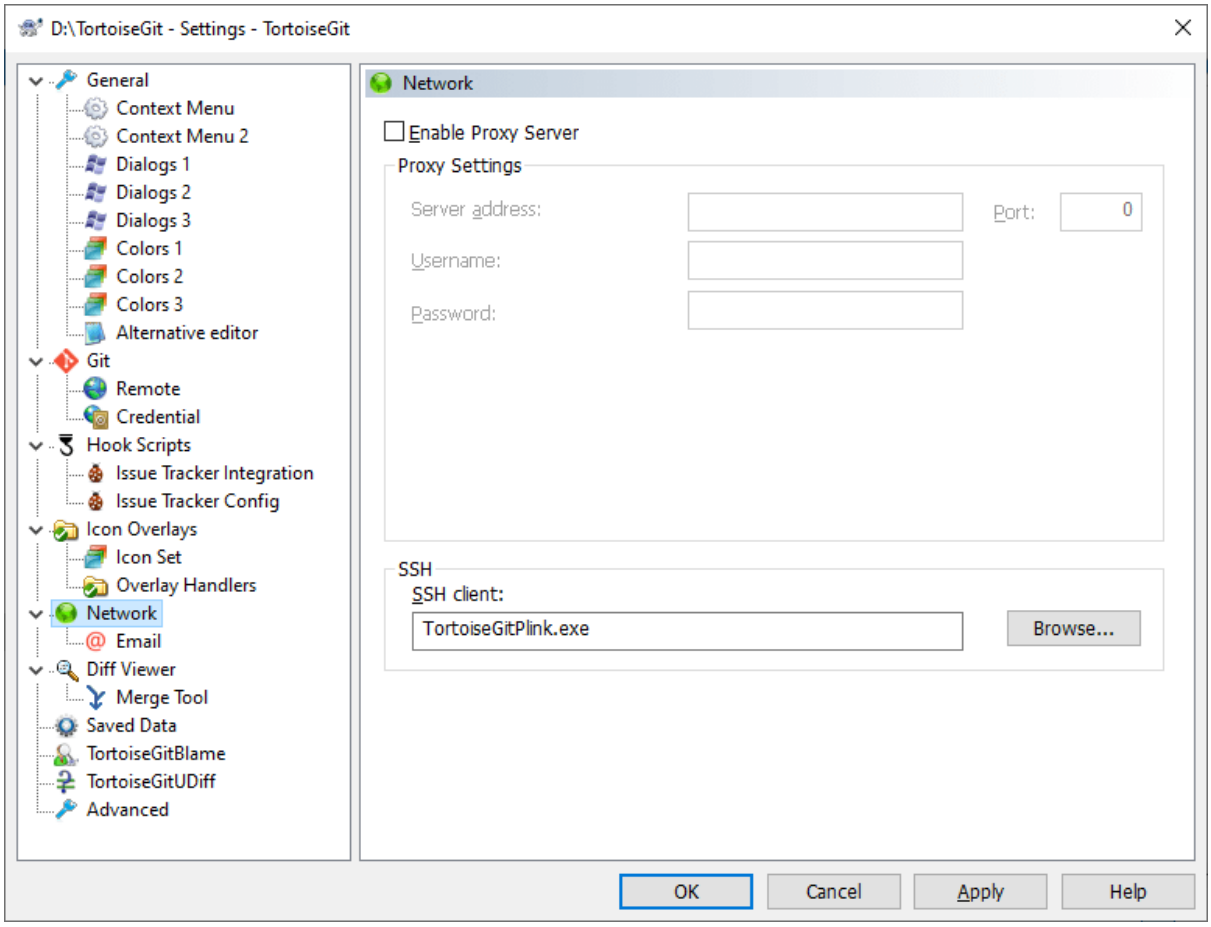


Figure 2.86. The Settings Dialog, Network Page

Here you can configure your proxy server, if you need one to get through your company's firewall.

The proxy server settings here do only affect Git for Windows (i.e., HTTP and HTTPS protocols). If you are using OpenSSH/PuTTY/Tortoise(Git)Plink you have to set up the proxy server settings there separately. In order to do this, you need the main PuTTY tool, which is not shipped with TortoiseGit. Preferably you store the proxy settings to the "Default Settings" configuration there, so that it is applied by default.

If you need to set up per-repository proxy settings, you will need to use the Git `config` file to configure this. Consult Section G.3.29, “git-config(1)” for more details.

You can also specify which program TortoiseGit should use to establish a secure connection to a git repository which is access using SSH. We recommend that you use TortoiseGitPlink.exe. This is a version of the popular Plink program, and is included with TortoiseGit, but it is compiled as a Windowless app, so you don't get a DOS box popping up every time you authenticate.

You must specify the full path to the executable. For TortoiseGitPlink.exe this is the standard TortoiseGit bin directory. Use the Browse button to help locate it, e.g.:

```
"C:\Program Files\TortoiseGit\bin\TortoiseGitPlink.exe"
```

Tip

If you want to use OpenSSH shipped by Git for Windows/msysGit just enter `ssh.exe`.

One side-effect of not having a window is that there is nowhere for any error messages to go, so if authentication fails you will simply get a message saying something like “Unable to write to standard output”. For this reason we

recommend that you first set up using standard Plink. When everything is working, you can use TortoiseGitPlink with exactly the same parameters.

TortoiseGitPlink does not have any documentation of its own because it is just a minor variant of Plink. Find out about command line parameters from the *PuTTY website* [https://www.chiark.greenend.org.uk/~sgtatham/putty/]

To avoid being prompted for a password repeatedly, you might also consider using a password caching tool such as Pageant. This is also available for download from the PuTTY website or included in the TortoiseGit package. (Also see Section 2.1.5, "Authentication".)

Finally, setting up SSH on clients is a non-trivial process which is beyond the scope of this help file. However, you can find a guide in the TortoiseGit FAQ listed under Appendix F, *Tips and tricks for SSH/PuTTY*.

2.36.3.1. Email settings

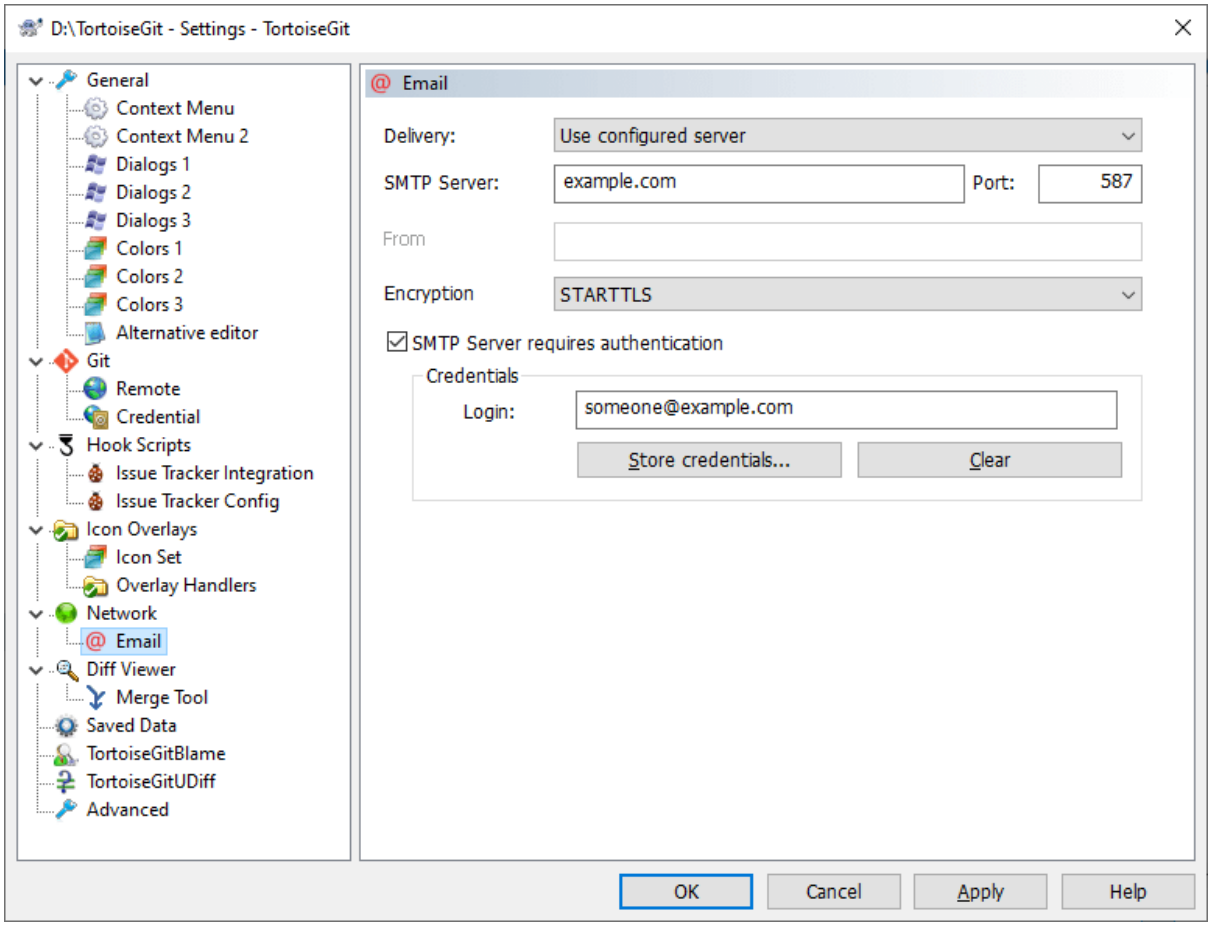


Figure 2.87. The Settings Dialog, email settings

This page allows you to specify configure how mails should be send.

SMTP, directly to destination server

When this option is selected, TortoiseGit directly connects to the SMTP server(s) (on port 25) which is/are responsible for the specific destination email-address(es). This is the default for TortoiseGit (unless some different method is configured).

Important

This might be problematic if your ISP blocks outgoing SMTP connections (port 25) or you have a dial-up internet connection. In the ladder case some destination MTAs might not accept your mails or mark them as SPAM.

MAPI

When this option is selected, TortoiseGit uses the Microsoft Messaging API (MAPI) for sending mails. For this, you need a MAPI capable mail client (e.g. Thunderbird or Outlook).

> **Important**
>
> If you don't send patches as attachments, you might need to make sure that no auto line wrapping takes place. For Thunderbird there is an add-on (*Toggle Word Wrap* [https://addons.mozilla.org/de/thunderbird/addon/toggle-word-wrap/]) available.

use configured server

This is the recommended way for sending mails. Just enter the same data as in your mail tools (MUA).

2.36.4. External Program Settings

Here you can define your own programs that TortoiseGit should use. The default setting is to use tools which are installed alongside TortoiseGit.

Read Section 2.18.6, "External Diff/Merge Tools" for a list of some of the external diff/merge programs that people are using with TortoiseGit.

2.36.4.1. Diff Viewer

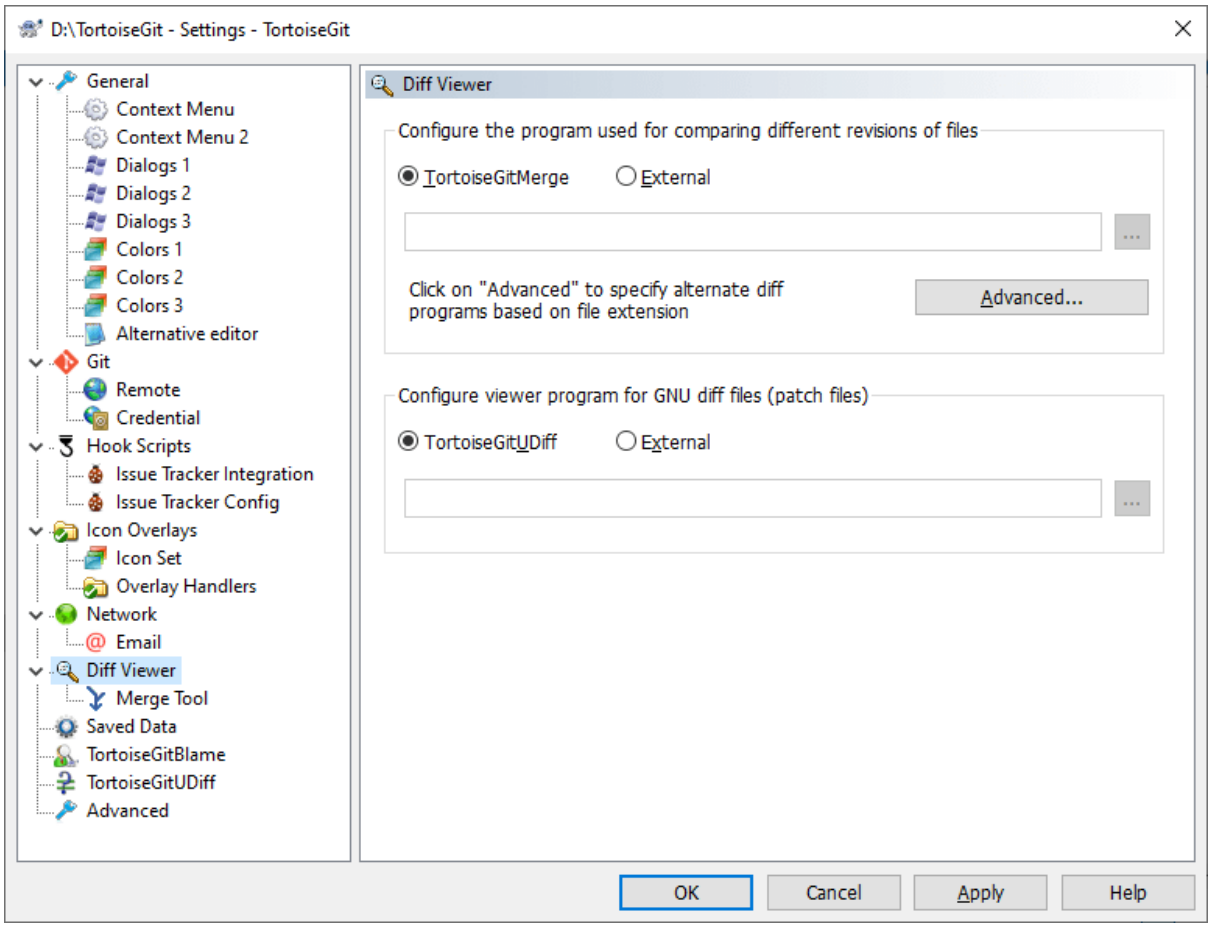


Figure 2.88. The Settings Dialog, Diff Viewer Page

An external diff program may be used for comparing different revisions of files. The external program will need to obtain the filenames from the command line, along with any other command line options. TortoiseGit uses

substitution parameters prefixed with `%`. When it encounters one of these it will substitute the appropriate value. The order of the parameters will depend on the Diff program you use.

%base

The original file without your changes

%bname

The window title for the base file

%mine

Your own file, with your changes

%yname

The window title for your file

%bpath

Full path to the original file

%ypath

Full path to your file

%brev

The revision of the original file, if available

%yrev

The revision of the second file, if available

%wtroot

Path to the working tree

The window titles are not pure filenames. TortoiseGit treats that as a name to display and creates the names accordingly. So e.g. if you're doing a diff from a file in revision 123 with a file in your working tree, the names will be `filename: revision 123` and `filename: working tree`

For example, with ExamDiff Pro:

```
C:\Path-To\ExamDiff.exe %base %mine --left_display_name:%bname --
right_display_name:%yname
```

or with KDiff3:

```
C:\Path-To\kdiff3.exe %base %mine --L1 %bname --L2 %yname
```

or with WinMerge:

```
C:\Path-To\WinMerge.exe -e -ub -dl %bname -dr %yname %base %mine
```

or with Araxis:

```
C:\Path-To\compare.exe /max /wait /title1:%bname /title2:%yname %base %mine
```

If you have configured an alternate diff tool, you can access TortoiseGitMerge *and* the third party tool from the context menus. Context menu → Diff uses the primary diff tool, and **Shift**+ Context menu → Diff uses the secondary diff tool.

2.36.4.1.1. Unified-Diff/GNU Diff/Patch File Viewer

Instead of TortoiseGitUDiff an external viewer program for unified-diff files (GNU diff or patch files) may be used. Basically, there is no parameter required - the file name if the unified diff file to be opened will be appended automatically. If you need to pass it as a different parameter the substitution `%1` can be used. There also is the parameter substitution `%title` available for passing the title to be shown in the title bar (i.e., meta data of the diff).

For example, with Notepad2 (shipped with TortoiseGit):

```
Notepad2.exe /s "Diff Files" /t %title
```

or (equivalent)

```
Notepad2.exe /s "Diff Files" /t %title %1
```

If you have configured an alternate unified diff tool, you can access TortoiseGitUDiff *and* the third party tool from the context menus. When you hold the **Shift**-key while opening the context menu the secondary unified diff tool is started.

2.36.4.2. Merge Tool

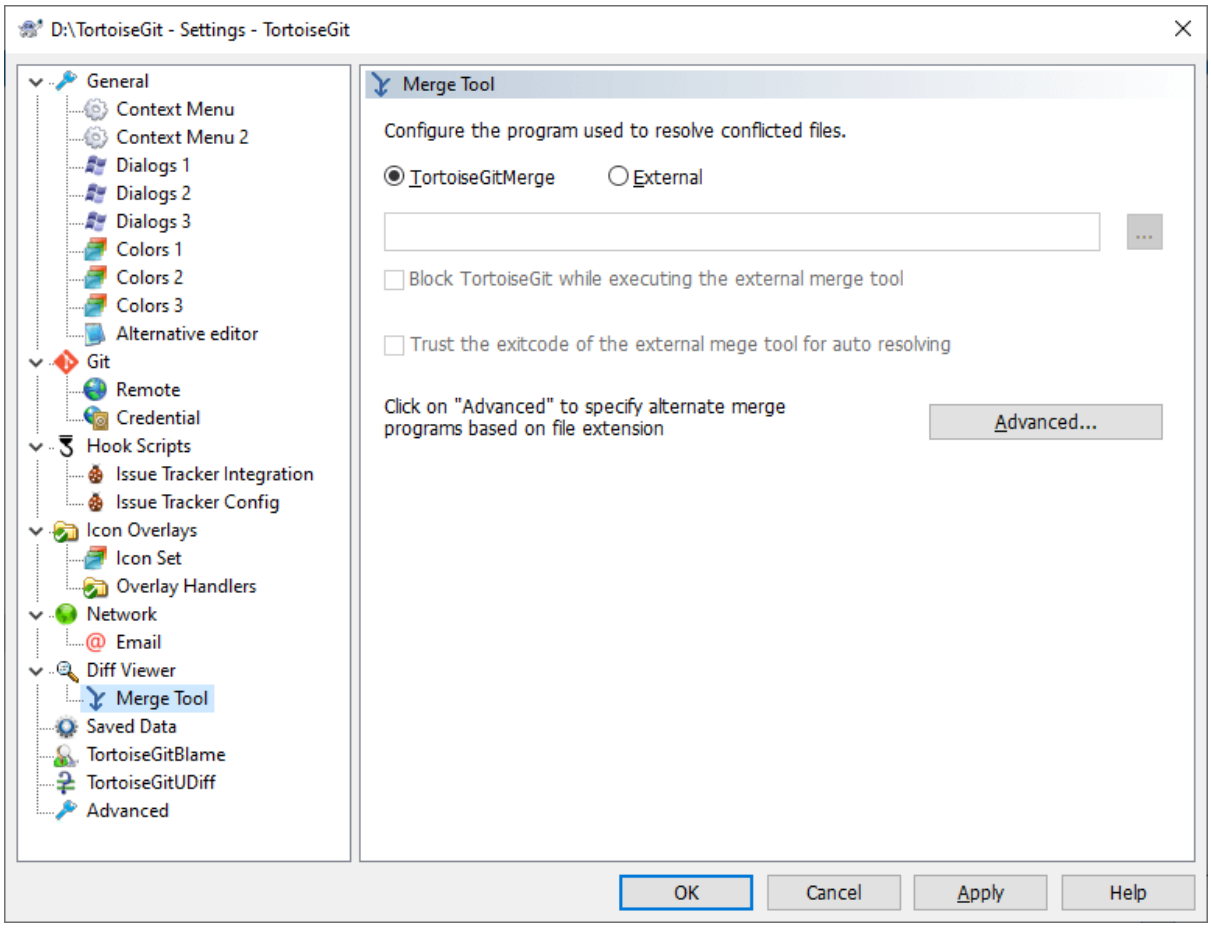


Figure 2.89. The Settings Dialog, Merge Tool Page

An external merge program used to resolve conflicted files. Parameter substitution is used in the same way as with the Diff Program.

%base

the original file without your or the others changes

%bname

The window title for the base file

%mine

your own file, with your changes

%yname

The window title for your file

%theirs

the file as it is in the repository

%tname

The window title for the file in the repository

%merged

the conflicted file, the result of the merge operation

%mname

The window title for the merged file

%wtroot

Path to the working tree

For example, with Perforce Merge:

```
C:\Path-To\P4Merge.exe %base %theirs %mine %merged
```

or with KDiff3:

```
C:\Path-To\kdiff3.exe %base %mine %theirs -o %merged --L1 %bname --L2
 %yname --L3 %tname
```

or with Araxis:

```
C:\Path-To\compare.exe /max /wait /3 /title1:%tname /title2:%bname /title3:
%yname %theirs %base %mine %merged /a2
```

or with WinMerge (2.12 or later):

```
C:\Path-To\WinMergeU.exe /e /ub /fr /wl /wm /dl %bname /dm %tname /dr
 %yname  %base %theirs %mine /o %merged /ar
```

TortoiseGit creates temporary files with similar file names as the conflicted file (`CONFLICTED.BASE.EXT`, `CONFLICTED.LOCAL.EXT` and `CONFLICTED.REMOTE.EXT`). These files are automatically removed when the conflict is marked as resolved using TortoiseGit, TortoiseGitMerge, or TortoiseGitIDiff.

When using an external tool, a conflicted file needs to be marked as revolved in TortoiseGit manually (doing so also removes the temporary files). This can be simplified and might also be automated: TortoiseGit can be configured to synchronously executing the merge tool (Block TortoiseGit while executing the external merge tool). Then TortoiseGit waits until the external merge tool is closed and asks whether to resolve the conflict (the temporary files are removed in any case). If the external merge tool provides a proper exit code (0 for success) you can trust the exit code to automatically mark the conflicted file as resolved (as Git does, cf. Section G.3.88, “git-mergetool(1)”).

2.36.4.3. Diff/Merge Advanced Settings

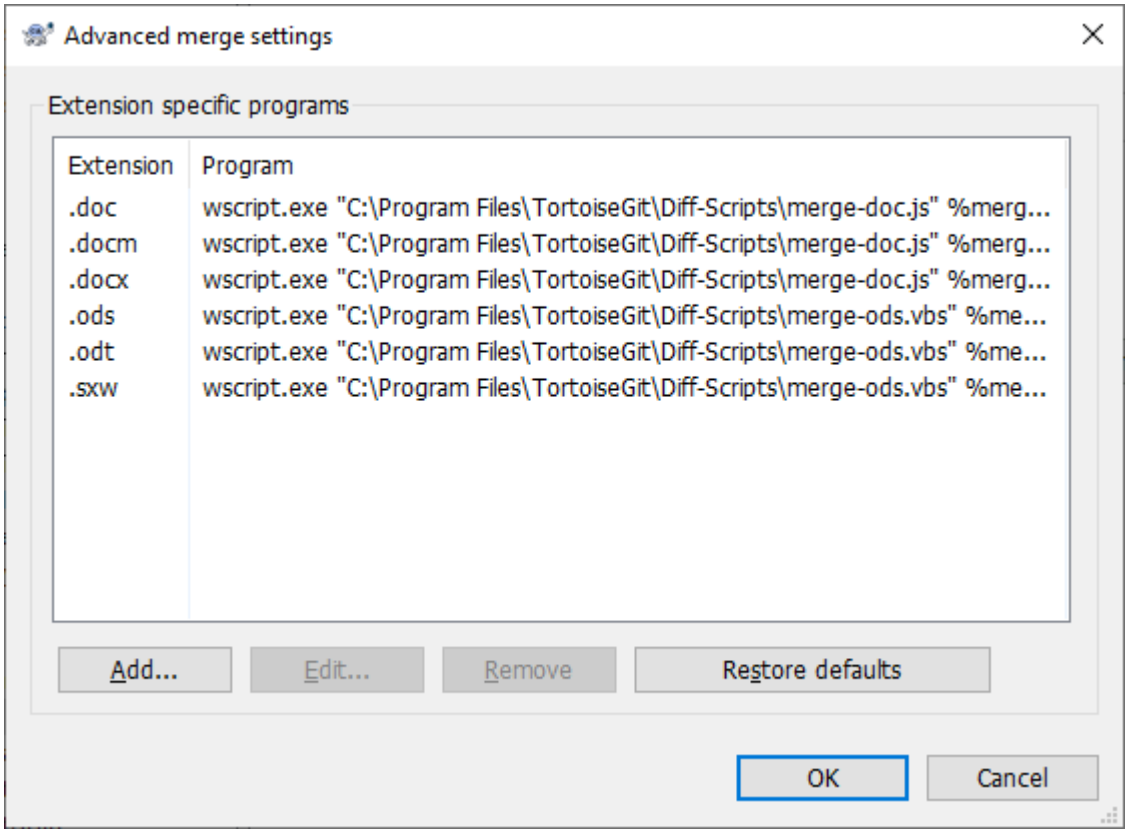


Figure 2.90. The Settings Dialog, Diff/Merge Advanced Dialog

In the advanced settings, you can define a different diff and merge program for every file extension. For instance you could associate Photoshop as the “Diff” Program for `.jpg` files :-)

To associate using a file extension, you need to specify the extension. Use `.bmp` to describe Windows bitmap files.

2.36.4.4. Alternative editor

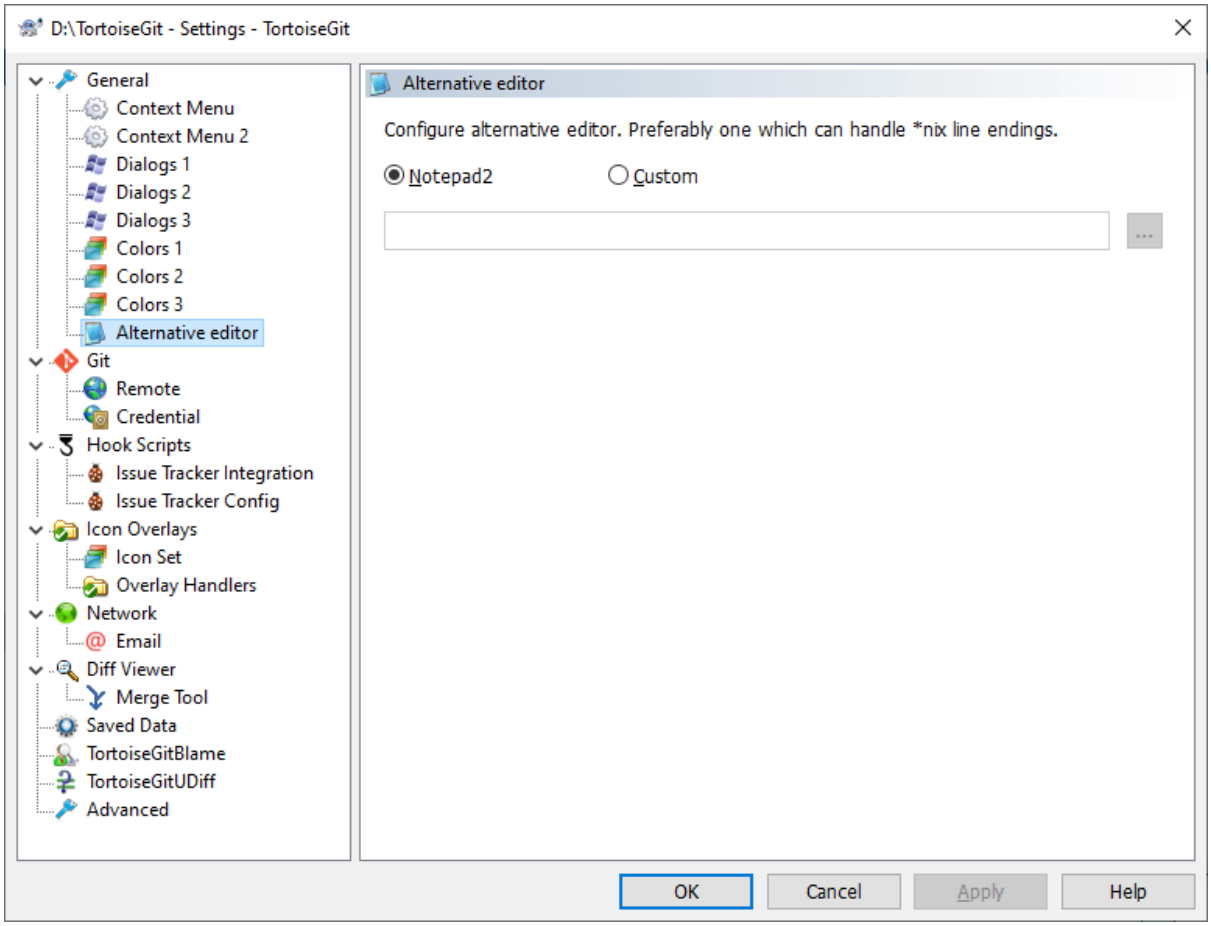


Figure 2.91. The Settings Dialog, Alternative editor Page

The original Windows Notepad program does not behave well on files which do not have standard CR-LF line-endings. However, a lot of git configuration files do not have a standard CR-LF line-ending. Because of this TortoiseGit uses a free (shipped) Notepad replacement *Notepad2* [https://www.flos-freeware.ch/notepad2.html] which displays the line-endings correctly by default.

2.36.5. Saved Data Settings

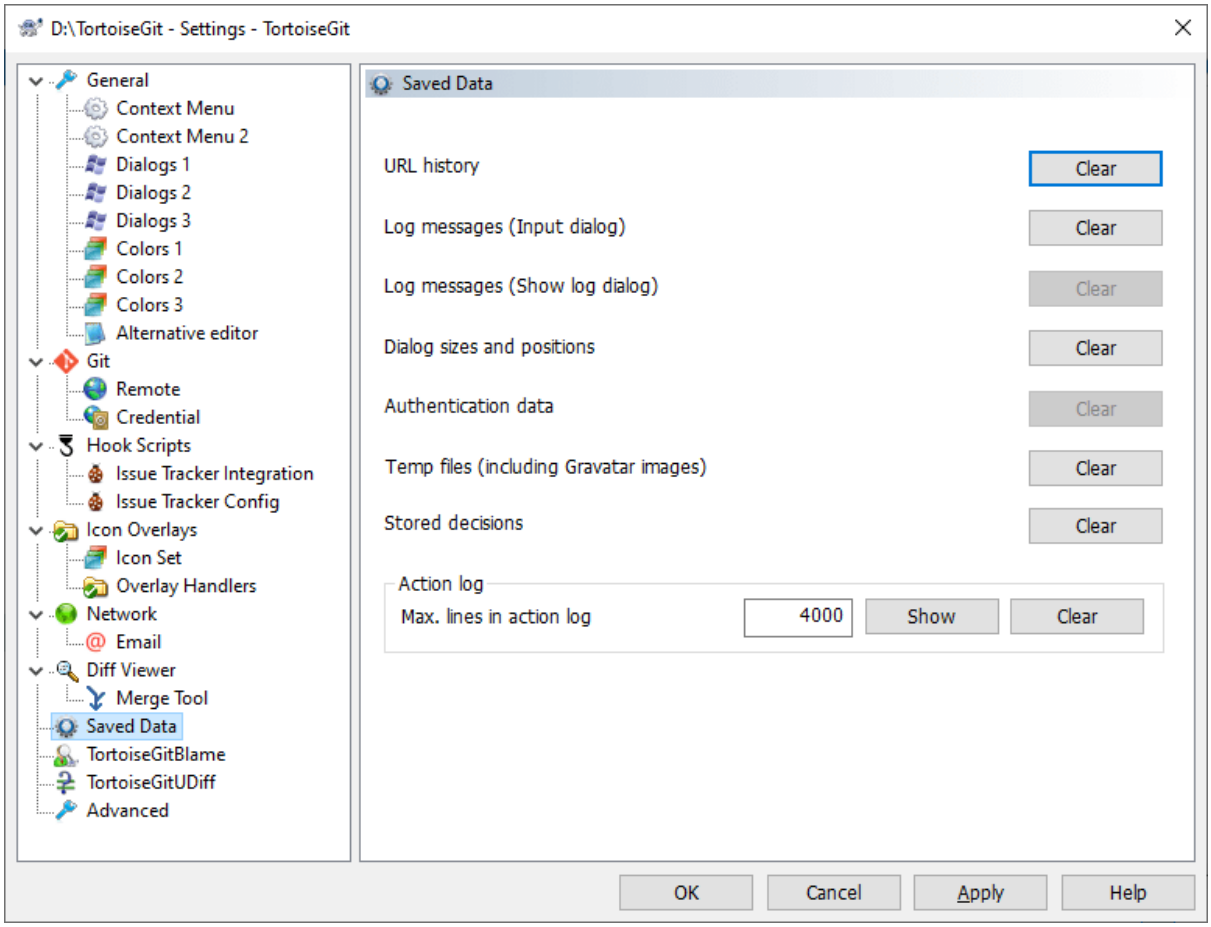


Figure 2.92. The Settings Dialog, Saved Data Page

For your convenience, TortoiseGit saves many of the settings you use, and remembers where you have been lately. If you want to clear out that cache of data, you can do it here.

URL history

Whenever you checkout a working tree, merge changes or use the repository browser, TortoiseGit keeps a record of recently used URLs and offers them in a combo box. Sometimes that list gets cluttered with outdated URLs so it is useful to flush it out periodically.

If you want to remove a single item from one of the combo boxes you can do that in-place. Just click on the arrow to drop the combo box down, move the mouse over the item you want to remove and type **Shift+Del**.

Log messages (Input dialog)

TortoiseGit stores recent commit log messages that you enter. These are stored per repository, so if you access many repositories this list can grow quite large.

Log messages (Show log dialog)

TortoiseGit caches log messages fetched by the Show Log dialog to save time when you next show the log. If someone else edits a log message and you already have that message cached, you will not see the change until you clear the cache. Log message caching is enabled on the Log Cache tab.

Dialog sizes and positions

Many dialogs remember the size and screen position that you last used.

Action log

TortoiseGit keeps a log of everything written to its progress dialogs. This can be useful when, for example, you want to check what happened in a recent update command.

The log file is limited in length and when it grows too big the oldest content is discarded. By default 4000 lines are kept, but you can customize that number.

From here you can view the log file content, and also clear it.

The log file is located at `%LOCALAPPDATA%\TortoiseGit\logfile.txt`.

2.36.6. Git

2.36.6.1. The hierarchical Git configuration

Git uses the concept of a hierarchical configuration (cf. Section G.3.29, “git-config(1)”). I.e. there are multiple levels; settings in higher levels override values in lower levels. The Effective tab shows you the effective values for the current scope (read-only).

Select any level (e.g. Local - the current repository settings stored locally in `.git/config`, Project - settings for the current repository stored within the repository in `/.tgitconfig`, Global - settings for the current user, System - settings for all users of the system) to see the values stored there.

In order to change settings select a level, enter the values, select where to store to and click on Apply.

Caution

If you want to inherit a value of a higher level don't leave a textbox empty (this means than an empty string will be stored, which might evaluate to `true`), select Inherit instead.

2.36.6.2. Git Config

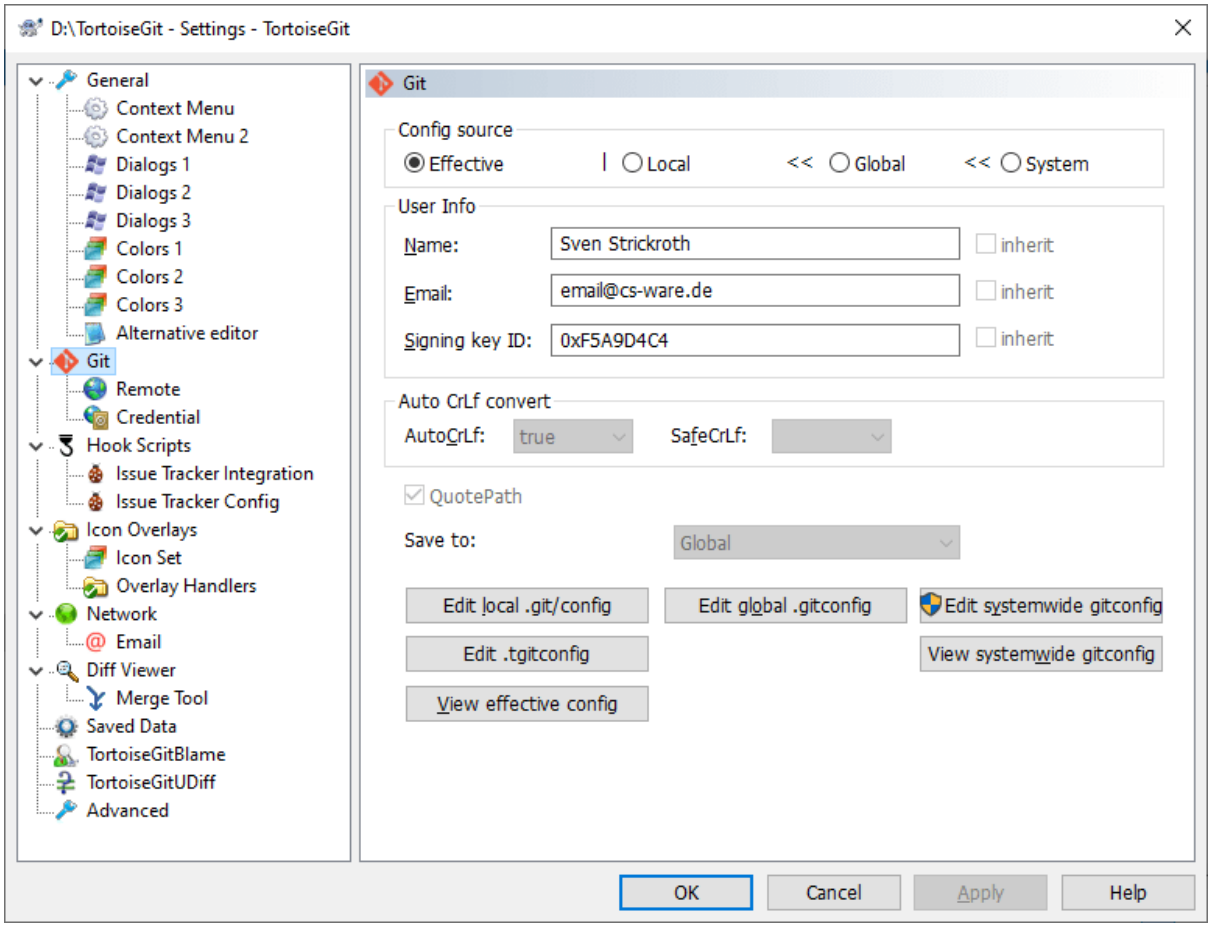


Figure 2.93. The Settings Dialog, Git

Set git basic configuration

Name and Email are required for git to operate correctly.

AutoCrLf If true, makes git convert CRLF at the end of lines in text files to LF when reading from the filesystem, and convert in reverse when writing to the filesystem. The variable can be set to input, in which case the conversion happens only while reading from the filesystem but files are written out with LF at the end of lines. A file is considered "text" (i.e. be subjected to the AutoCrLf mechanism) based on the file's CRLF attribute, or if CRLF is unspecified, based on the file's contents.

SafeCrLf If true, makes git check if converting CRLF as controlled by `core.autocrlf` is reversible. Git will verify if a command modifies a file in the work tree either directly or indirectly. For example, committing a file followed by checking out the same file should yield the original file in the work tree. If this is not the case for the current setting of `core.autocrlf`, git will reject the file. The variable can be set to "warn", in which case git will only warn about an irreversible conversion but continue the operation.

QuotePath Controls the `core.quotepath` setting which might be interesting when you have non ASCII filenames: See Section G.3.29, “git-config(1)”.

Important

If you have problems entering/storing data please see Section 2.36.6.1, “The hierarchical Git configuration”.

2.36.6.3. Remote

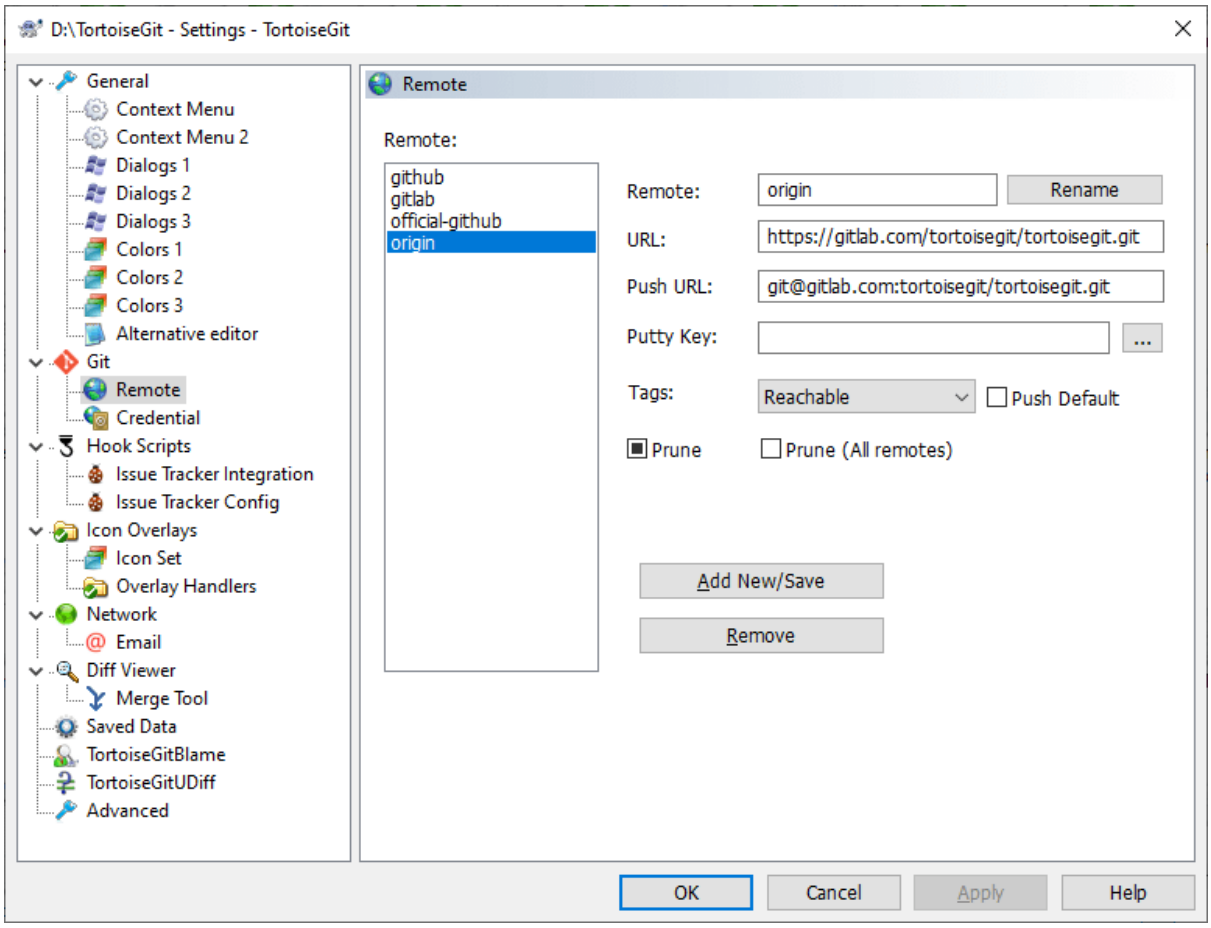


Figure 2.94. The Settings Dialog, Git, Remote

Set Git remote configuration

Remote The name of the remote, usually the default one is called `origin`.

URL The URL of the remote. It can be HTTP / HTTPS / SSH / Git protocol or local file system.

Push URL The Push URL of the remote. It is for some cases you cannot use the same URL to fetch and push (for example, fetch via password-less Git protocol but push via SSH). Otherwise, leave it empty. Note: This is not designed for forking workflow. For forking workflow, you should have 2 remotes. The format is the same as URL.

Putty Key The putty key file to load when performing network operations.

Tag This sets `remote.<name>.tagopt` config, which controls the default tag fetching behavior of the specified remote. `Reachable`: Download tags that are reachable from remote branch heads (default behavior). `None`: No tags are downloaded (`--no-tags`). (git 1.9 and later) `All`: All tags as well as branches are downloaded (`--tags`). (prior to git 1.9) `All tags only`: Only all tags are downloaded but no branches are downloaded (`--tags`). Use case of `All`: Always fetch tags from a git-svn mirror. Subversion tags never exist on trunk, so such tags are not reachable from branch heads.

Push Default Selecting this means to always push to this remote (cf. Section G.3.29, “git-config(1)”) Default is false.

Prune This sets `remote.<name>.prune` config, which controls the default prune option of remote tracking branches of the specified remote. Default is false.

2.36.6.4. Credential

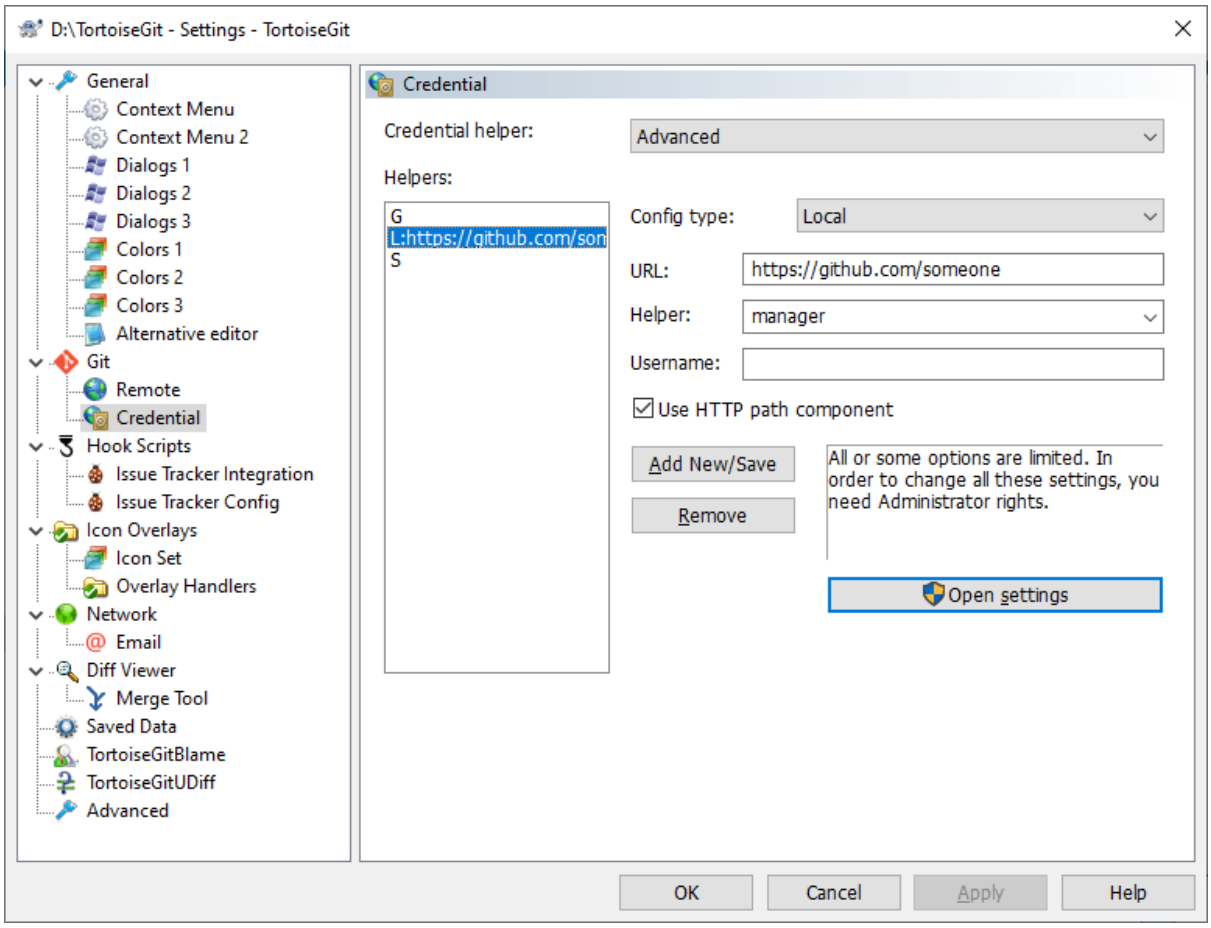


Figure 2.95. The Settings Dialog, Git, Credential

Set simple credential helper configuration

Advanced This is used if the credential helper configuration does not match any simple settings. If you choose other than `Advanced`, except the corresponding `credential.helper`, all other config keys `credential.*` or `credential.*.*` are removed.

None No credential config keys are in all config levels.

manager-core - this repository only Git Credential Manager Core (manager-core; https://github.com/microsoft/Git-Credential-Manager-Core) is enabled in local config only. This option is visible only if manager-core is installed.

manager - this repository only Git Credential Manager (manager; https://github.com/microsoft/Git-Credential-Manager-for-Windows) is enabled in local config only. This option is visible only if manager is installed.

wincred - this repository only wincred is enabled in local config only. This option is visible only if wincred is installed.

winstore - this repository only winstore is enabled in local config only. This option is visible only if winstore is installed for current Windows user.

manager-core - current Windows user Git Credential Manager Core (manager-core; https://github.com/microsoft/Git-Credential-Manager-Core) is enabled in global config only. This option is visible only if manager-core is installed.

manager - current Windows user Git Credential Manager (manager; https://github.com/microsoft/Git-Credential-Manager-for-Windows) is enabled in global config only. This option is visible only if manager is installed.

wincred - current Windows user wincred is enabled in global config only. This option is visible only if wincred is installed.

winstore - current Windows user winstore is enabled in global config only. This option is visible only if winstore is installed for current Windows user.

manager-core - all Windows users Git Credential Manager Core (manager-core; https://github.com/microsoft/Git-Credential-Manager-Core) is enabled in system config only. This option is visible only if manager-core is installed. Change to this option requires administrator privileges.

manager - all Windows users Git Credential Manager (manager; https://github.com/microsoft/Git-Credential-Manager-for-Windows) is enabled in system config only. This option is visible only if manager is installed. Change to this option requires administrator privileges.

wincred - all Windows users wincred is enabled in system config only. This option is visible only if wincred is installed. Change to this option requires administrator privileges.

Advanced credential helper configuration

Config type Either Local, Global or System config.

URL Define a context-specific configuration based on URL pattern. By default, the path component is not considered as a different context.

Helper Select a credential helper program. manager-core, manager, wincred, and winstore are predefined in TortoiseGit. It is possible to use other credential helpers or with extra options.

Username A default username, if one is not provided in the URL.

Use HTTP path component Also considers the path component of URL to match the configuration context.

You can find more information at Section G.4.3, “gitcredentials(7)”.

2.36.7. Client Side Hook Scripts

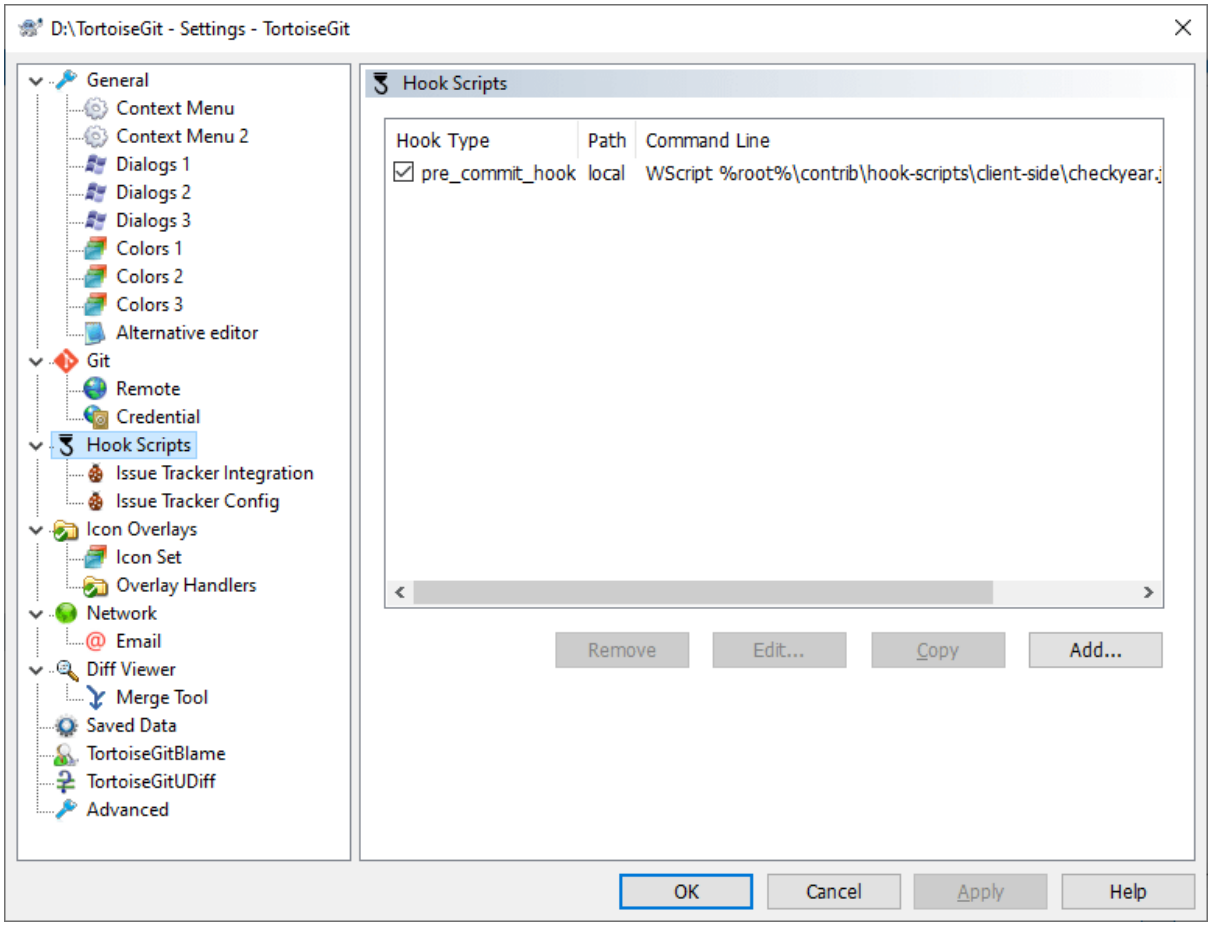


Figure 2.96. The Settings Dialog, Hook Scripts Page

This dialog allows you to set up hook scripts which will be executed automatically when certain TortoiseGit actions are performed on the client side.

For various security and implementation reasons, hook scripts are defined locally on a machine, rather than as project properties. You define what happens, no matter what someone else commits to the repository. Of course you can always choose to call a script which is itself under version control.

One application for such hooks might be to call a program like `GitWCRev.exe` (Chapter 3, *The GitWCRev Program*) to update version numbers after a commit, and perhaps to trigger a rebuild.

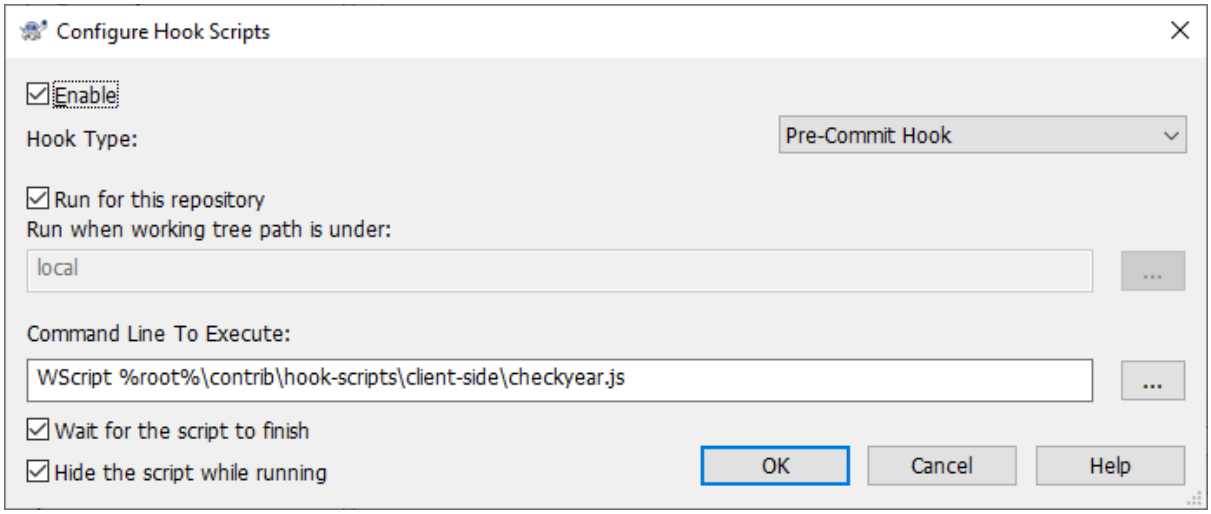


Figure 2.97. The Settings Dialog, Configure Hook Scripts

To add a new hook script, simply click Add and fill in the details.

There are currently six types of hook script available

Start-commit

Called before the commit dialog is shown. You might want to use this if the hook modifies a versioned file and affects the list of files that need to be committed and/or commit message. However you should note that because the hook is called at an early stage, the full list of objects selected for commit is not available.

Pre-commit

Called after the user clicks OK in the commit dialog, and before the actual commit begins. This hook has a list of exactly what will be committed.

Post-commit

Called after the commit finished successfully.

Pre-push

Called before actual Git push begins.

Post-push

Called after pushing finishes (whether successful or not).

Pre-rebase

Called before rebasing starts (after clicking on Start or autostart).

A hook is defined for a particular working tree path. You only need to specify the top level path; if you perform an operation in a sub-folder, TortoiseGit will automatically search upwards for a matching path. Use `*` for matching all working trees.

If the checkbox Run for this repository is checked then the hook script is attached to the current repository and configured automatically for every clone and checkout (the hook information is stored in the file `.tgitconfig` in the repository root so that it will be automatically shared with all other developers using TortoiseGit >= 2.7.1; for

security reasons TortoiseGit asks the user before running a hook which is configured and shared in the repository). In this case, you can specify paths for the command line with the replacement string `%root%` for the path to the working tree folder. The hook script has to be inside the repository and also be checked out of course (please also note the security implications below). If a user locally configures a hook for the exact repository root folder, the client side defined hook takes precedence.

Next you must specify the command line to execute, starting with the path to the hook script or executable. This could be a batch file, an executable file or any other file which has a valid windows file association, e.g. a perl script.

The command line includes several parameters which get filled in by TortoiseGit. The parameters passed depend upon which hook is called. Each hook has its own parameters which are passed in the following order:

Start-commit

```
PATH MESSAGEFILE CWD
```

Pre-commit

```
PATH MESSAGEFILE CWD
```

Post-commit

```
CWD (commit was amend (true or false))
```

Pre-push

```
ERROR CWD
```

Post-push

```
ERROR CWD
```

Pre-rebase

```
(upstream branch) (rebased branch) ERROR CWD
```

The meaning of each of these parameters is described here:

PATH

A path to a temporary file which contains all the paths for which the operation was started in UTF-8 encoding. Each path is on a separate line in the temp file.

MESSAGEFILE

Path to a file containing the log message for the commit. The file contains the text in UTF-8 encoding. After successful execution of the start-commit and pre-commit hooks, the log message is read back, giving the hook a chance to modify it.

ERROR

Path to a file containing the error message. If there was no error, the file will be empty.

CWD

The current working directory with which the script is run. This is set to the working tree root.

Note that although we have given these parameters names for convenience, you do not have to refer to those names in the hook settings. All parameters listed for a particular hook are always passed, whether you want them or not ;-)

If you want the Git operation to hold off until the hook has completed, check Wait for the script to finish.

Normally you will want to hide ugly DOS boxes when the script runs, so Hide the script while running is checked by default.

Caution

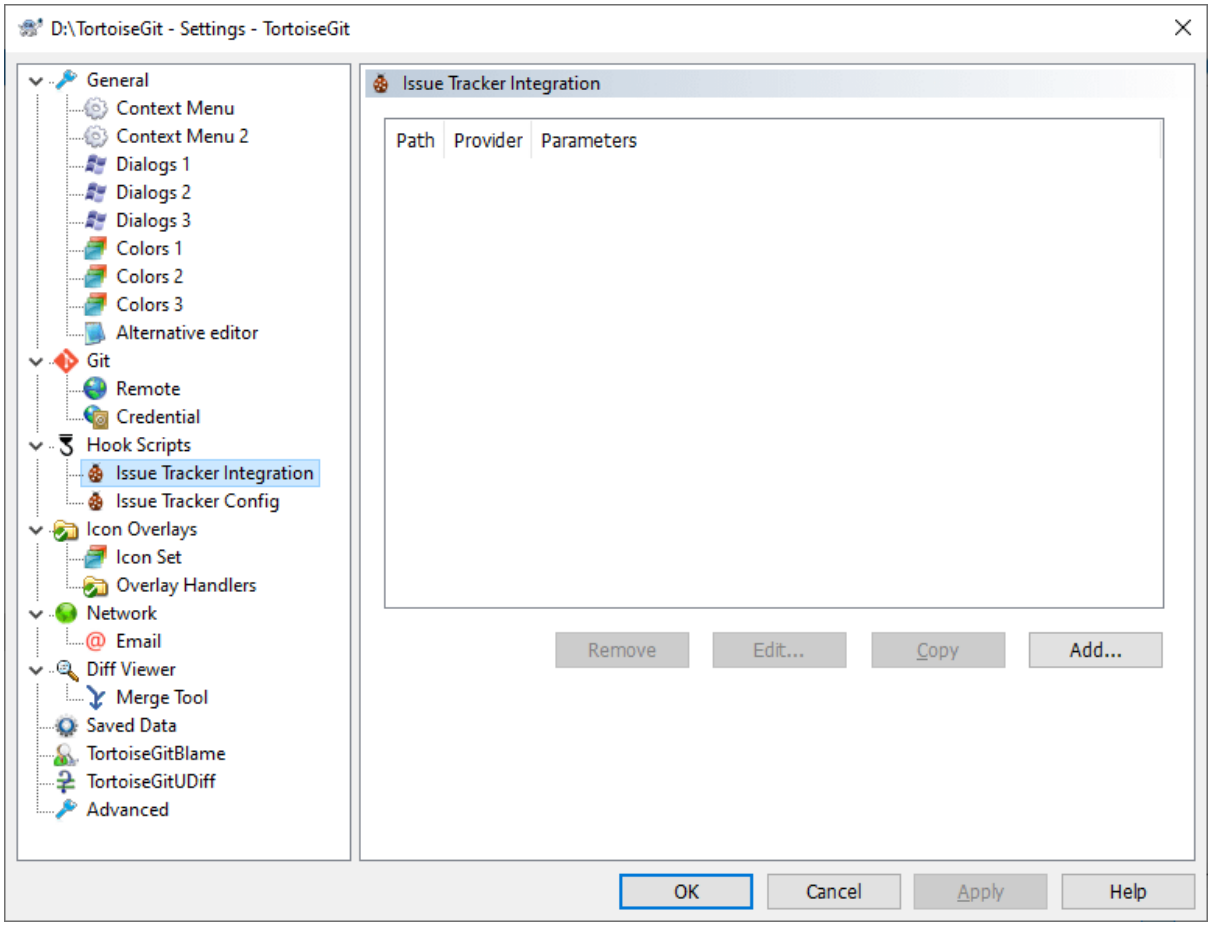
If you are executing a versioned file/script from the repository, please note that the file possibly gets altered by third parties unnoticed (e.g. after pull or merge).

2.36.7.1. Issue Tracker Integration

TortoiseGit can use a COM plugin to query issue trackers when in the commit dialog. The use of such plugins is described in Section 2.35.2, “Getting Information from the Issue Tracker”. If your system administrator has provided you with a plugin, which you have already installed and registered, this is the place to specify how it integrates with your working tree.

Tip

There is also a hierarchical git configuration to associate issue tracker plugin with your project, rather than with to a specific directory path. Such settings are more portable. See Section 2.35, “Integration with Bug Tracking Systems / Issue Trackers” to configure these settings.



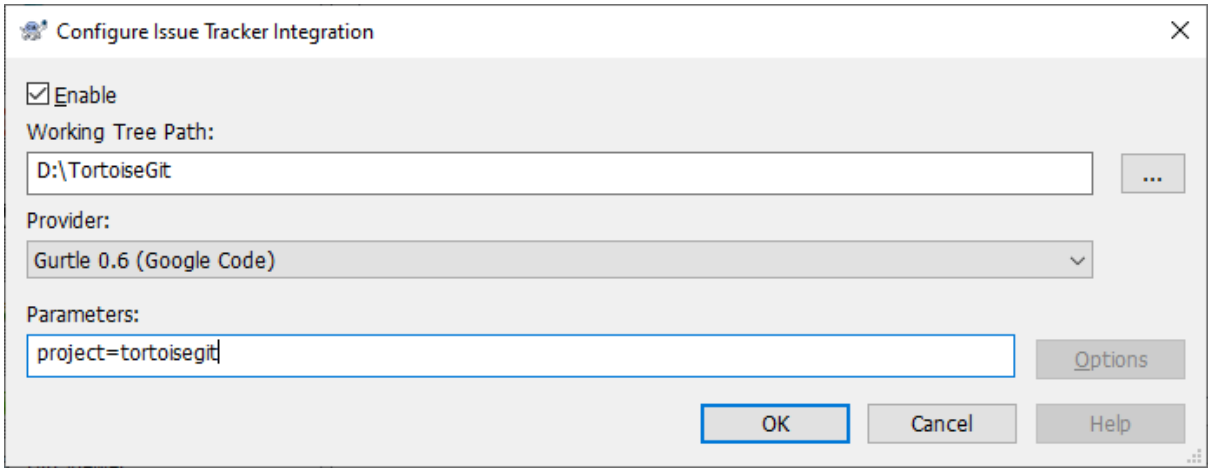


Figure 2.98. The Settings Dialog, Issue Tracker Integration Page

Click on Add... to use the plugin with a particular working tree. Here you can specify the working tree path, choose which plugin to use from a drop down list of all registered issue tracker plugins, and any parameters to pass. The parameters will be specific to the plugin, but might include your user name on the issue tracker so that the plugin can query for issues which are assigned to you.

2.36.7.2. Config

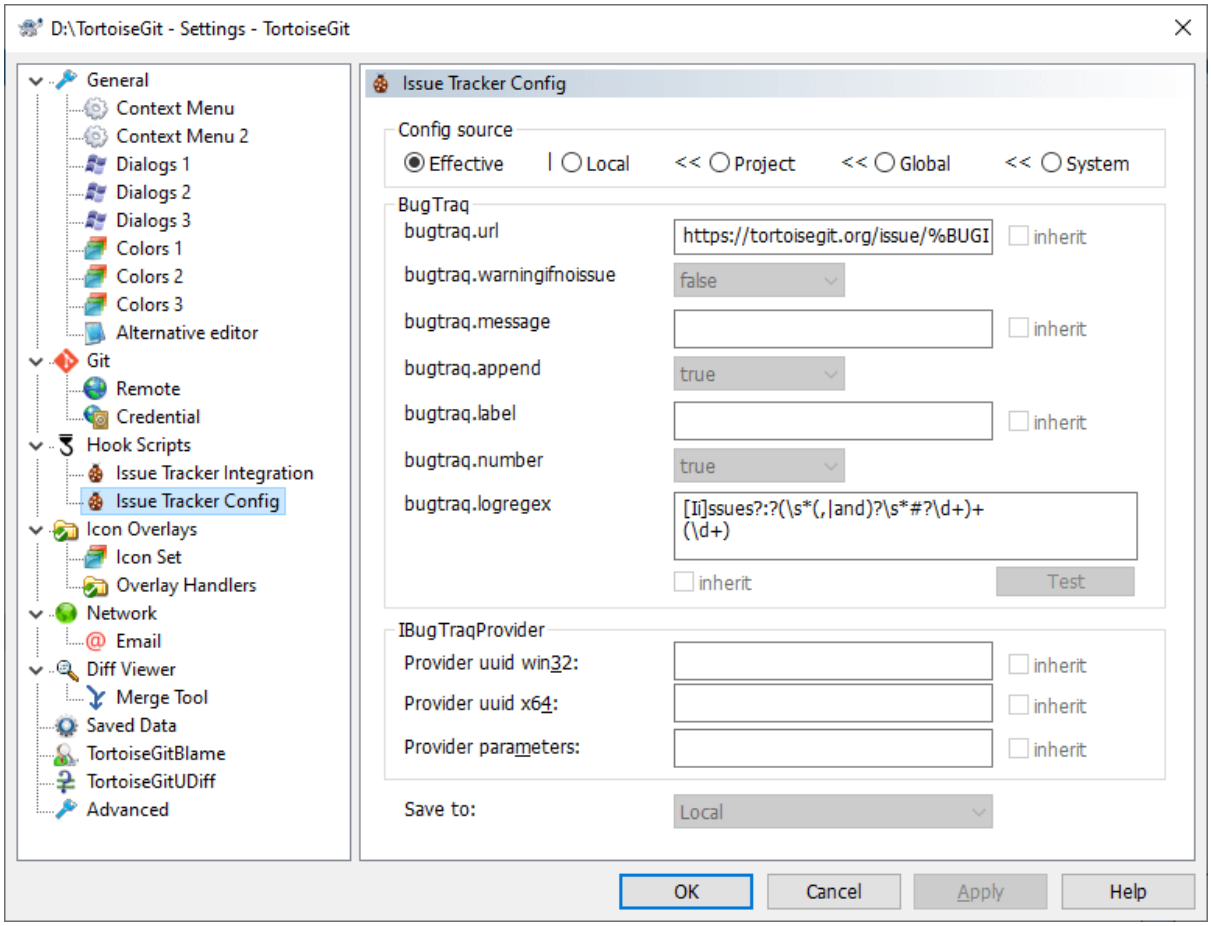


Figure 2.99. The Settings Dialog, Issue Tracker Config

See Section 2.35, “Integration with Bug Tracking Systems / Issue Trackers” for a descriptions of the different options.

Important

If you have problems entering/storing data please see Section 2.36.6.1, “The hierarchical Git configuration”.

2.36.8. TortoiseGitBlame Settings

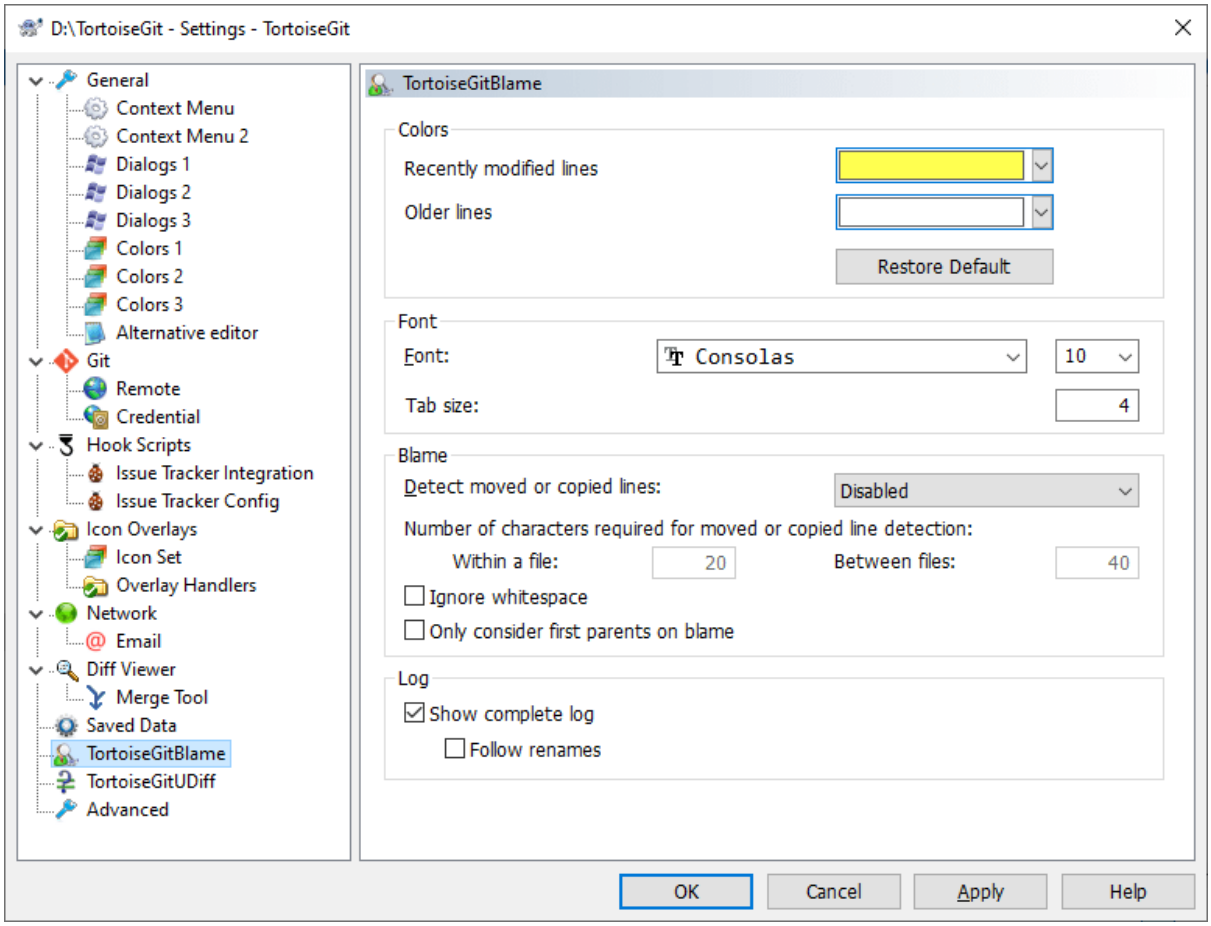


Figure 2.100. The Settings Dialog, TortoiseGitBlame Page

The settings used by TortoiseGitBlame are controlled from the main context menu, not directly with TortoiseGitBlame itself. Details for the parameters for the blame algorithm are described in Section G.3.9, “git-blame(1)”.

Colors

TortoiseGitBlame can use the background color to indicate the age of lines in a file. You set the endpoints by specifying the colors for the newest and oldest revisions, and TortoiseGitBlame uses a linear interpolation between these colors according to the repository revision indicated for each line.

Font

You can select the font used to display the text, and the point size to use. This applies both to the file content, and to the author and revision information shown in the left pane.

Tab size

Defines how many spaces to use for expansion when a tab character is found in the file content.

Detect moved or copied lines

Disabled Traditional blame algorithm, the search for parents is limited to the file and will follow renames.

Within file Extra passes of inspection are applied to detect moved and copied lines within the file (`git blame -M`).

From modified files In addition to the annotated file detect moved or copied lines from all modified files within a commit (`git blame -C`).

At file creation In addition to the annotated file and the modified files within a commit detect moved or copied lines from other files in the commit that creates the file (`git blame -C -C`).

From existing files In addition detect moved or modified lines from other files in any commit (`git blame -C -C -C`).

Number of characters required for moved or copied line detection

Lower bound on the number of alphanumeric characters that Git must detect as moving/copying between files for it to associate those lines with the parent commit.

Within a file Number of alphanumeric characters required to detect moving lines within a file (`git blame -M|<num>|`).

Between files Number of alphanumeric characters required to detect moved or copied lines between files (`git blame -C|<num>|`).

Ignore whitespace

Defines if whitespace is ignored when comparing the parent's version and the child's version to find where the lines came from (`git blame -w`).

Show complete log

Defines if the log should be complete, i.e. the log contains all changes for a file, even the changes have no impact on the file content of the annotated revision. If deactivated the log contains only revisions which last modified a line for the annotated revision.

Follow renames

Defines if the log should follow renames, i.e. if the log does not stop when a file was renamed in the past, but include all changes before the rename.

2.36.9. TortoiseGitUDiff Settings

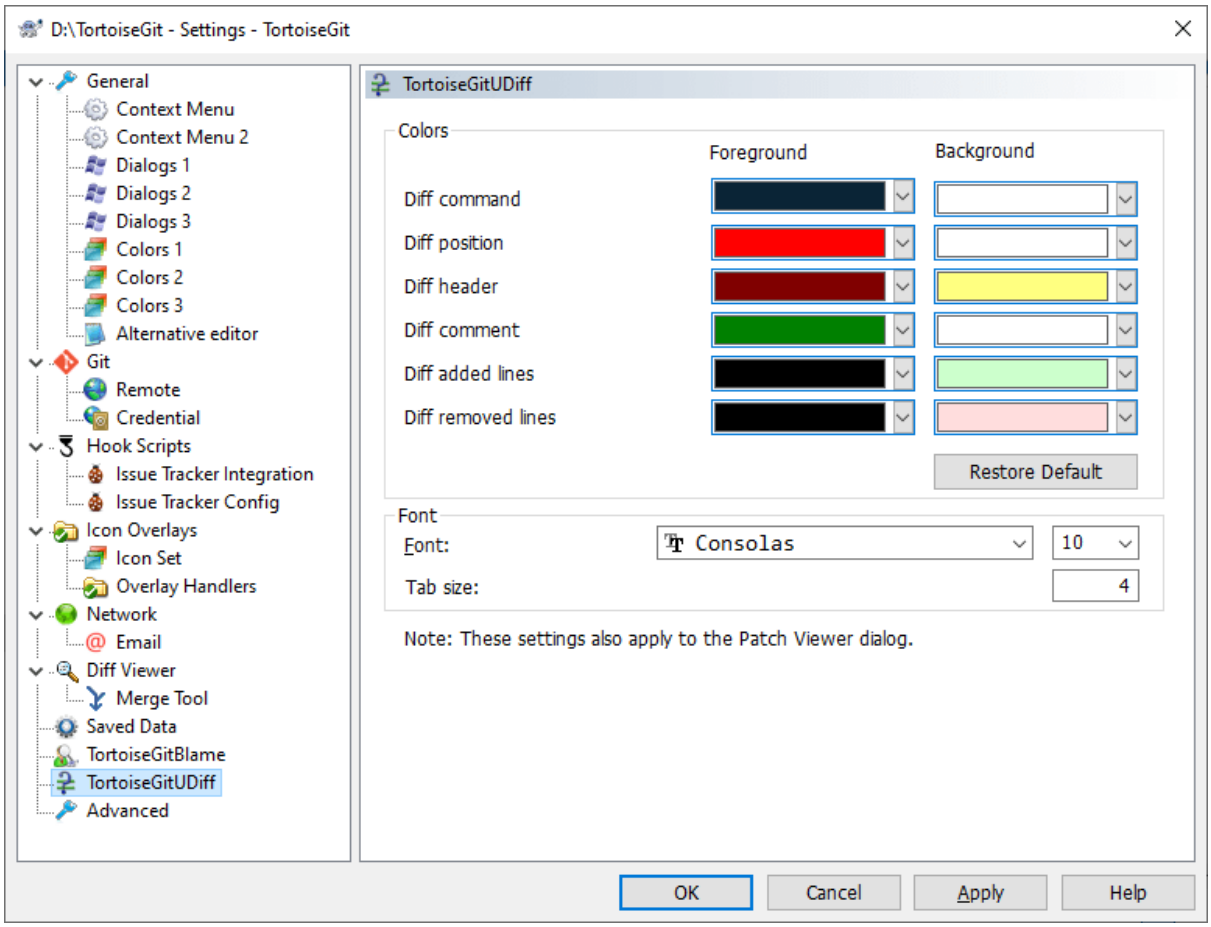


Figure 2.101. The Settings Dialog, TortoiseGitUDiff Page

The settings used by TortoiseGitUDiff are controlled from the main context menu, not directly with TortoiseGitUDiff itself.

Colors

The default colors used by TortoiseGitUDiff are usually a good choice, but you can configure them here.

Font

You can select the font used to display the text, and the point size to use.

Tabs

Defines how many spaces to use for expansion when a tab character is found in the file diff.

To select whether you would like to use the build-in or any alternative diff viewer program go to Section 2.36.4, “External Program Settings” preferences section in the leftward tree.

2.36.10. Advanced Settings

A few infrequently used settings are available only in the advanced page of the settings dialog. These settings modify the registry directly and you have to know what each of these settings is used for and what it does. Do not modify these settings unless you are sure you need to change them.

AutoCompleteMinChars

The minimum amount of chars from which the editor shows an auto-completion popup. The default value is 3.

AutocompleteParseMaxSize

The auto-completion list shown in the commit message editor can parse source code files and displays methods and variable names. This limits files to be parsed by their size in bytes. The default value is `300000`.

AutocompleteParseUnversioned

The auto-completion list shown in the commit message editor can parse source code files and displays methods and variable names. By default only versioned files are parsed. Set this value to `true` in order to also parse unversioned files.

AutocompleteRemovesExtensions

The auto-completion list shown in the commit message editor displays the names of files listed for commit. To also include these names with extensions removed, set this value to `true`.

BlockStatus

If you don't want the explorer to update the status overlays while another TortoiseGit command is running (e.g. Update, Commit, ...) then set this value to `true`.

CacheTrayIcon

To add a cache tray icon for the TGitCache program, set this value to `true`. This is really only useful for developers as it allows you to terminate the program gracefully.

CacheSave

To disable loading and saving cache for the TGitCache program, set this value to `false`. This is useful if you do not want to write the cache to disk, which can be a large file. The default is `true`.

ConflictDontGuessBranchNames

When merging a conflict, TortoiseGit tries to find a friendly branch name for the context menu and for the title in TortoiseGitMerge to make merging easier. As Git does only stores the `MERGE_HEAD` as a commit hash, TortoiseGit has to guess the branch name (cf. *issue #3700* [https://tortoisegit.org/issue/3700]) which might be wrong if a commit has several branches. You can use this option to disable this heuristic. The default is `false`.

CygwinHack

This enables some workarounds which enables TortoiseGit to be used with Cygwin Git. Cygwin Git, however, is not officially supported by TortoiseGit. See Section 2.36.1, “General Settings” for more information. The default is `false`.

Debug

Set this to `true` if you want a dialog to pop up for every command showing the command line used to start TortoiseGitProc.exe.

DebugOutputString

Set this to `true` if you want TortoiseGit to print out debug messages during execution. The messages can be captured with special debugging tools only (like *Debug View* [https://docs.microsoft.com/sysinternals/downloads/debugview] from the SysInternals Suite).

DialogTitles

The default format (value of 0) of dialog titles is `url/path - name of dialog - TortoiseGit`. If you set this value to 1, the format changes to `relative url/path based on repository name - name of dialog - TortoiseGit`.

DiffSimilarityIndexThreshold

This setting controls which similarity index threshold is passed to git diff (as the value for the parameters `-M` and `-C` in per cent, cf. `--find-copies` in Section G.3.44, "git-diff(1)"). The default value is `50`. You can disable finding renamed and copied files by setting this to `0`, for only detecting exact renames use `100`. You might need to remove the cache files `tortoisegit.data` and `tortoisegit.index` in the `.git` folders after changing this value.

DownloadAnimation

When performing `git.exe` or remote operations TortoiseGit dialogs play an animation with a flying turtle. This setting allows to disable the playing of the animation by setting it to `false`. The default value is `true`.

FullRowSelect

The status list control which is used in various dialogs (e.g., commit, check-for-modifications, add, revert, ...) uses full row selection (i.e., if you select an entry, the full row is selected, not just the first column). This is fine, but the selected row then also covers the background image on the bottom right, which can look ugly. To disable full row select, set this value to `false`.

GroupTaskbarIconsPerRepo

This option determines how the Win7 taskbar icons of the various TortoiseGit dialogs and windows are grouped together.

1. The default value is `3`. With this setting, the icons are grouped together by application type per working tree. All dialogs from TortoiseGit of one working tree are grouped together, all windows from TortoiseGitMerge of one working tree are grouped together, ... For example, if you have a log dialog and a push dialog open for working tree `C:\A`, and a check-for-modifications dialog and a log dialog for working tree `C:\B`, then there are two application icon groups shown in the Win7 taskbar, one group for each working tree. But TortoiseGitMerge windows are not grouped together with TortoiseGit dialogs.

Figure 2.102. Taskbar with default grouping

2. If set to `4`, then the grouping works as with the setting set to `3`, except that TortoiseGit, TortoiseGitMerge, TortoiseGitBlame, TortoiseGitIDiff and TortoiseGitUDiff windows of one working tree are all grouped together. For example, if you have the log dialog open and then double click on a modified file, the opened TortoiseGitMerge diff window will be put in the same icon group on the taskbar as the log dialog icon.

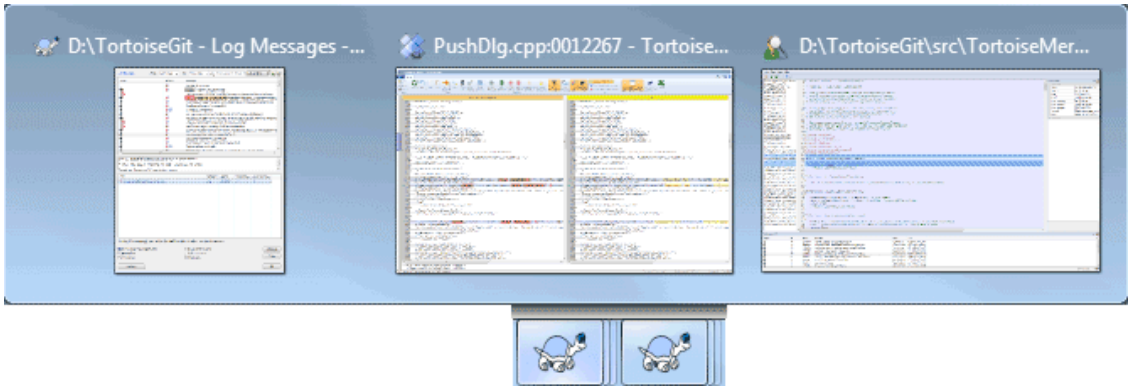


Figure 2.103. Taskbar with repository grouping

3. If set to `1`, then the grouping works as with the setting set to `3` (grouping by application), except that grouping takes place independently of the working tree. This was the default before TortoiseGit 1.8.1.2.

4. If set to `2`, then the grouping works as with the setting set to `4`, except that grouping takes place independently of the working tree. Thus all TortoiseGit icons are grouped to only show one icon.

GroupTaskbarIconsPerRepoOverlay

This has no effect if the option `GroupTaskbarIconsPerRepo` is set to `0` (see above).

If this option is set to `true`, then every icon on the Win7 taskbar shows a small colored rectangle overlay, indicating the working tree the dialogs/windows are used for.

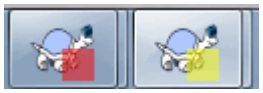


Figure 2.104. Taskbar grouping with repository color overlays

LogIncludeWorkingTreeChanges

This options controls whether the log dialog includes an entry for "Working Tree Changes". When using network drives (e.g., Samba), the log dialog might hang for big project because of large of files when calculating the working tree changes. Therefore, the possible expensive calculation can be disabled. The default is `true`.

LogShowSuperProjectSubmodulePointer

This option defines whether the commit of a submodule to which the super repository points to is highlighted with a branch like label (cf. *issue #2826* [https://tortoisegit.org/issue/2826]). The default is `true`.

LogTooManyItemsThreshold

In order to prevent delays displaying the files on a revision on the log dialog there is a maximum of items to be displayed enforced. The default is `1000`.

MaxRefHistoryItems

This options sets the maximum browse ref history (Right click ref hyperlink to find it). The default is `5`.

ModifyExplorerTitle

When using the status cache, the title bar of explorer windows are modified to include the branch name, stash count and if an upstream is set also the outgoing and incoming commits. Set this to `false` if you don't want this or if you have other tools which already do that. The default is `true`.

Msys2Hack

This enables some workarounds which enables TortoiseGit to be used with MSYS2 Git (do not enable this for the Git for Windows package!). MSYS2 Git, however, is not officially supported by TortoiseGit. See Section 2.36.1, “General Settings” for more information. The default is `false`.

NamedRemoteFetchAll

When set to `false`, `fetch` and `pull` don't fetch the default refspec for a named remote. The default is `true`.

NoSortLocalBranchesFirst

This option toggles if the branches are sorted fully by name (`true`) or if local branches should appear above remote ones (git default, `false`). The default value is `false`.

NumDiffWarning

If you want to show the diff at once for more items than specified with this settings, a warning dialog is shown first. The default is `10`.

OverlaysCaseSensitive

Starting with TortoiseGit 2.4.0 the overlay icons are case sensitive on filenames. The change was introduced to fix several issues related to casing (such as *issue #2654* [https://tortoisegit.org/issue/2654]) and git tools (such as `git log`) being case sensitive on paths. Upon *issue #2980* [https://tortoisegit.org/issue/2980] this is configurable starting from TortoiseGit 2.5.0, however, enabling is not recommended. The default is `true`.

ProgressDlgLinesLimit

The Git progress dialog shows the output of the executed `git.exe` commands. The number of lines are limited for performance reasons. The default is `50000`, minimum is `50`.

ReaddUnselectedAddedFilesAfterCommit

This option toggles the re-adding of unselected added files after a commit. Up to TortoiseGit 1.7.10 added files which were not checked on a commit, were removed from the index and unversioned after the commit. Set this value to `false` to restore the old behavior. Set this value to `true` to re-add these files again after the commit (default).

RefreshFileListAfterResolvingConflict

This option toggles whether the file lists of the commit dialog, resolve conflicts and rebase dialog automatically refresh when a conflict is marked as resolved. By default this is set to `true`, but in certain cases, e.g. when refreshing takes lots of time or you want to prevent the scrolling to the top, this can be set to `false`. However, then a manual refresh (e.g. by pressing **F5**) is necessary.

RememberFileListPosition

This option toggles whether the file lists of the add, commit, revert, resolve and rebase dialog remember the last selected line on a refresh. The default is `true`.

SanitizeCommitMsg

This option trims space, CR, LF characters at the end of commit messages you enter. This covers commit, rebase, notes, annotated tag. This value is `true` by default. If such trimming breaks your scripts/plugins, you can disable trimming by set it to `false`.

ScintillaDirect2D

This option enables the use of Direct2D accelerated drawing in the Scintilla control which is used as the edit box in e.g. the commit dialog (also for the attached patch window), the unified diff viewer and TortoiseGit-Blame. With some graphic cards, however, this sometimes doesn't work properly so that the cursor to enter text isn't always visible, the redraw does not work or the background is flashing. It's disabled by default. You can turn this feature on by setting this value to `true`.

ShellMenuAccelerators

TortoiseGit uses accelerators for its explorer context menu entries. Since this can lead to doubled accelerators (e.g. the `Git Commit` has the **Alt-C** accelerator, but so does the `Copy` entry of explorer). If you don't want or need the accelerators of the TortoiseGit entries, set this value to `false`.

ShortHashLengthForHyperLinkInLogMessage

The minimum length of commit hashes that TortoiseGit shows hyper-link for in log messages. Default is `8`.

ShowContextMenuIcons

This can be useful if you use something other than the windows explorer or if you get problems with the context menu displaying incorrectly. Set this value to `false` if you don't want TortoiseGit to show icons for the shell context menu items. Set this value to `true` to show the icons again.

ShowAppContextMenuIcons

If you don't want TortoiseGit to show icons for the context menus in its own dialogs, set this value to `false`.

ShowListBackgroundImage

If you do not want to have a small background image in list controls (e.g. Commit Dialog) set this value to `false`. Set this value to `true` to show the images again (default).

SquashDate

Using this setting you can control which date is used on squashing commits. Set this value to `1` if you want to use the date of the latest commit. Set this value to `2` if you want to use the current date. Set this value to `0` to use the date of the first commit (into which all others are squashed, default).

StyleCommitMessages

The commit and log dialog use styling (e.g. bold, italic) in commit messages (see Section 2.5.5, “Commit Log Messages” for details). If you don't want to do this, set the value to `false`.

StyleGitOutput

The Git.exe progress dialogs shows the output of a Git.exe process and use colors to highlights errors and warnings. If you don't want to do this, set the value to `false`.

TGitCacheCheckContentMaxSize

TGitCache checks the content of files by hashing them and comparing the SHA1 in order to calculate the file statuses if the timestamps (to index) mismatch. This option allows to restrict this behavior for files which do not exceed a specific size (in KiB). The default maximum file size is 10 MiB (i.e., 10 * 1024 KiB = `10240` KiB). Set this to 0 in order to make TGitCache only check the timestamps (as TortoiseGit 1.7.0 up to 1.7.12 did; before TortoiseGit 1.9.0.0 this was controlled by `TGitCacheCheckContent`). Disabling checking the file contents can lower disk access and CPU time of the TGitCache process, however, overlay accuracy might not be as accurate as with checking of the file contents enabled.

UseCustomWordBreak

The standard edit controls do not stop on forward slashes like they're found in paths and URLs. TortoiseGit uses a custom word break procedure for the edit controls. If you don't want that and use the default instead, set this value to 0. If you only want the default for edit controls in combo boxes, set this value to 1.

UseLibgit2

This makes TortoiseGit to use libgit2 as much as possible (e.g. for adding files to the index). If you do not want TortoiseGit to use libgit2 for file operations, set this value to `false`.

VersionCheck

TortoiseGit checks whether there's a new version available about once a week. If you don't want TortoiseGit to do this check, set this value to `false`.

VersionCheckPreview

Set this to `true` to make TortoiseGit also check for new preview releases. The default in all stable releases is `false`.

Win8SpellChecker

Set this to `true` to make TortoiseGit use the Windows 8+ spell checker (cf. Section 1.4.4, “Spell checker”). The default is `false`.

2.36.11. Exporting TortoiseGit Settings

If you want to export all your client settings to use on another computer you can do so using the Windows registry editor `regedt32.exe`. Go to the registry key `HKCU\Software\TortoiseGit` and export it to a reg file. On the other computer, just import that file again (usually, a double click on the reg file will do that).

Remember to save Git's general settings, which you can find in the Git configuration file `.gitconfig` and/or the folder `.config/git` which both are located in your user profile directory.

2.37. Working with worktrees

A worktree can be seen as an additional checkout of a Git repository (cf. [def_working_tree]).

Use the TortoiseGit→Worktrees option to manage worktrees. When selecting this option, a dialog pops up where all current worktrees are listed. The first entry is the main repository. It cannot be deleted because it contains the `.git`-folder with branches. All other worktrees use that `.git`-folder for book keeping. To remove/lock/unlock a worktree, right-click it and select the desired action.

2.37.1. Creating a worktree

To create a new worktree, specify a folder. By default for the path the main repo path is selected. Modify that to your liking. For example from `C:\Users\jdoe\TortoiseGit` to `C:\Users\jdoe\another-work-tree`. To specify which revision is checked out, select HEAD, a branch, tag, or a concrete commit. A branch can only be checked out at a single worktree. Alternatively use Force to check out the branch twice, but this is not recommended.

Check Section G.3.159, “git-worktree(1)” for further details.

2.38. git svn dcommit

Commit each diff from a specified HEAD directly to the SVN repository, and then rebase or reset (depending on whether or not there is a diff between SVN and HEAD). This will create a revision in SVN for each commit in git. It is recommended that you run git-svn fetch and rebase (not pull or merge) your commits against the latest changes in the SVN repository.

If you need/want to use `--use-log-author` or `--add-author-from`, please set those in git config (cf. Section G.3.29, “git-config(1)”), also see issue #2824 [https://tortoisegit.org/issue/2824].

Git Style Commit (--rmdir): Remove directories from the SVN tree if there are no files left behind. SVN can version empty directories, and they are not removed by default if there are no files left in them. git cannot version empty directories. Enabling this flag will make the commit to SVN act like git.

You can find more information at Section G.3.142, “git-svn(1)”.

2.39. Git LFS Locking

This section talks about how to use the Git LFS (Large File Support) locking feature.

File locking allows developers to lock files they are updating to prevent other users from updating them at the same time. Concurrent edits in Git repositories will lead to merge conflicts, which are very difficult to resolve in large binary files.

2.39.1. Setting up the repository

To use Git LFS locking you need to set up a repository. Currently this step needs to be done through Git command line.

Git has built-in tools for resolving merge conflicts in text files (such as source code, documentation, etc.). The first step to use File Locking is to define what file types need the extra overhead. The Git LFS track command includes a `--lockable` flag. The following command will store `*.jpg` files in LFS, and mark them as lockable:

```
$ git lfs track "*.jpg" --lockable
```

This adds the following line to your `.gitattributes` file:

```
*.jpg filter=lfs diff=lfs merge=lfs -text lockable
```

If you'd like to register a file type as lockable, without using LFS, you can edit the `.gitattributes` file directly:

```
*.yml lockable
```

Once file patterns in `.gitattributes` are lockable, Git LFS will make them read-only on the local file system automatically. This prevents users from accidentally editing a file without locking it first.

2.39.2. Locking a file

In order to edit a lockable file you need to lock it using context menu via TortoiseGit → LFS → Lock.

This is the client-server command and, thus, requires access to a remote server.

2.39.3. Unlocking a file

When you're done editing a file it's good practice to push and unlock it as soon as possible. To unlock the file use the context menu and choose TortoiseGit → LFS → Unlock.

This is the client-server command and, thus, requires access to a remote server.

2.39.4. Show Locks Dialog

In order to open Locks Dialog use TortoiseGit → LFS → Locks in any folder inside the working tree.

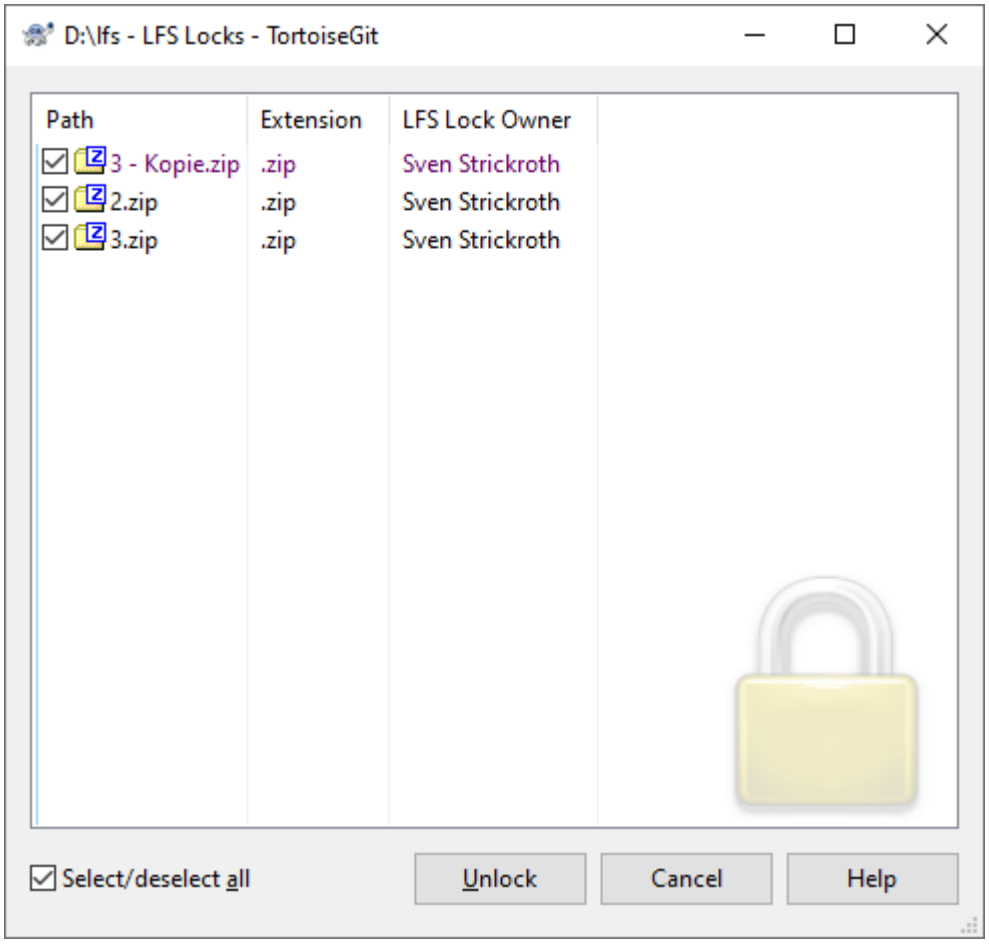


Figure 2.105. Locks Dialog

You can use this dialog to see all locks in the project. Also, you can unlock files here. If unlocking fails you will be offered to use force unlock.

2.40. Final Step

Donate! **[https://tortoisegit.org/donate]**

Even though TortoiseGit and TortoiseGitMerge are free, you can support the developers by sending in patches and play an active role in the development. You can also help to cheer us up during the endless hours we spend in front of our computers.

Please also have a look at the list of people who contributed to the project by sending in patches or translations.

Chapter 3. The GitWCRev Program

GitWCRev is Windows console program which can be used to read the status of a Git working tree or specific files inside a working tree and optionally perform keyword substitution in a template file - another alternative could be git filters (cf. Section G.4.2, “gitattributes(5)”). This is often used as part of the build process as a means of incorporating working tree information into the object you are building. Typically it might be used to include the revision number in an “About” box.

3.1. The GitWCRev Command Line

GitWCRev reads the Git status of all files in a working tree OR a directory/file inside a working tree including submodules. It records the HEAD commit revision and the commit timestamp, it also records whether there are local modifications in the passed path. The status revision and modification status are displayed on stdout.

GitWCRev.exe is called from the command line or a script, and is controlled using the command line parameters.

```
GitWCRev WorkingTreePath [SrcVersionFile DstVersionFile] [-mMuUsdq]
```

`WorkingTreePath` is the path to the working tree OR a directory/file inside a working tree to check. The path may be absolute or relative to the current working directory.

If you want GitWCRev to perform keyword substitution, so that fields like repository revision is saved to a text file, you need to supply a template file `SrcVersionFile` and an output file `DstVersionFile` which contains the substituted version of the template.

You can specify ignore patterns for GitWCRev to prevent specific files and paths from being considered. The patterns are read from a file named `.gitwcrevignore`. The file is read from the working tree root. If the file does not exist, no files or paths are ignored. The `.gitwcrevignore` file can contain multiple patterns, separated by newlines. The patterns are matched against the paths relative to the repository root . For example, to ignore all files in the `/doc` folder of the TortoiseGit working tree, the `.gitwcrevignore` would contain the following lines:

```
/doc
/doc/*
```

To ignore all images, the ignore patterns could be set like this:

```
*.png
*.jpg
*.ico
*.bmp
```

Important

The ignore patterns are case-sensitive, just like Git is.

Tip

To create a file with a starting dot in the Windows explorer, enter `.gitwcrevignore.`. Note the trailing dot.

There are a number of optional switches which affect the way GitWCRev works. If you use more than one, they must be specified as a single group, e.g. `-sU`, not `-s -U`.

Switch	Description
-m	If this switch is given, GitWCRev will exit with `ERRORLEVEL 7` if the passed path contains local modifications. This may be used to prevent building with uncommitted changes present.
-M	Same as above, but includes the status of submodules.
-u	If this switch is given, GitWCRev will exit with `ERRORLEVEL 11` if the passed path contains unversioned items that are not ignored.
-U	Same as above, but includes the status of submodules.
-d	If this switch is given, GitWCRev will exit with `ERRORLEVEL 9` if the destination file already exists.
-s	If this switch is given, GitWCRev will exclude submodules. The default behavior is to also check submodules.
-F	If this switch is given, GitWCRev will ignore any `.gitwcrevignore` files and include all files.
-q	If this switch is given, GitWCRev will perform the keyword substitution without showing working tree status on stdout.

Table 3.1. List of available command line switches

If there is no error, GitWCRev returns zero. But in case an error occurs, the error message is written to stderr and shown in the console. And the returned error codes are:

Error Code	Description
1	Syntax error. One or more command line parameters are invalid.
2	The file or folder specified on the command line was not found.
3	The input file could not be opened, or the target file could not be created.
4	Could not allocate memory. This could happen if e.g. the source file is too big.
5	The source file can not be scanned properly.
6	Git error: libgit2 returned with an error when GitWCRev tried to find the information from the working tree.
7	The working tree has local modifications. This requires the `-m` or `-M` switch.
9	The output file already exists. This requires the `-d` switch.
10	The specified path is neither a working tree, a bare repository nor part of one.
11	The passed path has unversioned files or folders in it. This requires the `-u` or `-U` switch.

Table 3.2. List of GitWCRev error codes

3.2. Keyword Substitution

If a source and destination files are supplied, GitWCRev copies source to destination, performing keyword substitution as follows:

Keyword	Description
$WCREV$	Replaced with the HEAD commit revision of the working tree.
$WCREV=n$	Replaced with the HEAD commit revision of the working tree, trimmed to `n` chars. For example: `$WCREV=7$`
$WCBRANCH$	Replaced with the name of the current branch (or SHA-1 if HEAD is detached) of the working tree.
$WCLOGCOUNT$, $WCLOGCOUNT&$,	Replaced with the number of first-parent commits from HEAD back to the first commit. This number is guaranteed to increase

Keyword	Description
$WCLOGCOUNT+$, $WCLOGCOUNT-$	with every commit on the very same branch as long no history is rewritten and can be used as part of a version number (see *https://gcc.gnu.org/ml/gcc/2015-08/msg00148.html* [https://gcc.gnu.org/ml/gcc/2015-08/msg00148.html] and *https://gitlab.com/tortoisegit/tortoisegit/merge_requests/1* [https://gitlab.com/tortoisegit/tortoisegit/merge_requests/1] for more details). The `$WCLOGCOUNT&$`, `$WCLOGCOUNT+$` and `$WCLOGCOUNT-$` can be used to `AND`, add or subtract a predefined number from/to the number of commits. For example `$WCLOGCOUNT&65535$` will ensure the number is not exceeding 16 bit.
$WCDATE$, $WCDATEUTC$	Replaced with the commit date/time of the highest commit revision. By default, international format is used: `yyyy-mm-dd hh:mm:ss`. Alternatively, you can specify a custom format which will be used with `strftime()`, for example: `$WCDATE=%a %b %d %I:%M:%S %p$`. For a list of available formatting characters, look at the *online reference* [https://msdn.microsoft.com/en-us/library/fe06s4ak.aspx].
$WCNOW$, $WCNOWUTC$	Replaced with the current system date/time. This can be used to indicate the build time. Time formatting can be used as described for `$WCDATE$`.
$WCMODS$	`$WCMODS?TText:FText$` is replaced with `TText` if there are local modifications in the passed path, or `FText` if not. This will also evaluate to `true` if a submodule is checked out at a different commit (requires submodules not to be ignored).
$WCFILEMODS$	`$WCFILEMODS?TText:FText$` is replaced with `TText` if there are local modifications in the passed path, or `FText` if not. This does not check the checked out commit of submodules.
$WCUNVER$	`$WCUNVER?TText:FText$` is replaced with `TText` if there are unversioned items in the passed path, or `FText` if not.
$WCISTAGGED$	`$WCISTAGGED?TText:FText$` is replaced with `TText` if the HEAD commit is tagged, or `FText` if not.
$WCINGIT$	`$WCINGIT?TText:FText$` is replaced with `TText` if the entry is versioned, or `FText` if not. The result for directories is `false`, the only exception is the repository root.
$WCSUBMODULE$	`$WCSUBMODULE?TText:FText$` is replaced with `TText` if the passed path contains a submodules, or `FText` if not.
$WCSUBMODULEUP2DATE$	`$WCSUBMODULEUP2DATE?TText:FText$` is replaced with `TText` if all submodules are checked out at the version specified in the index of the parent working tree, or `FText` if not.
$WCMODSINSUBMODULE$	`$WCMODSINSUBMODULE?TText:FText$` is replaced with `TText` if a submodule contains uncommitted changes, or `FText` if not.
$WCUNVERINSUBMODULE$	`$WCUNVERINSUBMODULE?TText:FText$` is replaced with `TText` if a submodule contains unversioned items, or `FText` if not.
$WCMODSFULL$	`$WCMODSFULL?TText:FText$` combines is `$WCMODS$` and `$WCMODSINSUBMODULE$` and can be seen as a recursive check. replaced with `TText` if the passed path or any submodule under the passed path contains uncommitted changes, or `FText` if not.
$WCUNVERFULL$	`$WCUNVERFULL?TText:FText$` combines is `$WCUNVER$` and `$WCUNVERINSUBMODULE$` and can be seen as a recursive check. replaced with `TText` if the passed path or any submodule under the passed path contains unversioned items, or `FText` if not.

Table 3.3. List of available keywords

GitWCRev does not directly support nesting of expressions, so for example you cannot use an expression like:

```
#define SVN_REVISION    "$WCUNVER?$WCNOW$:$WCDATE$$"
```

But you can usually work around it by other means, for example:

```
#define DATE_NOW       $WCNOW$
#define DATE_COMMIT    $WCDATE$
#define DATE           "$WCUNVER?DATE_NOW:DATE_COMMIT$"
```

Tip

Some of these keywords apply to single files rather than to an entire working tree, so it only makes sense to use these when GitWCRev is called to scan a single file. This applies to `$WCINGIT$`.

3.3. Keyword Example

The example below shows how keywords in a template file are substituted in the output file.

```
// Test file for GitWCRev

char* Revision                    = "$WCREV$";
char* RevisionShort               = "$WCREV=7$";
char* Modified                    = "$WCMODS?Modified:Not modified$";
char* Unversioned                 = "$WCUNVER?Unversioned items found:no
 unversioned items$";
char* Date                        = "$WCDATE$";
char* DateUTC                     = "$WCDATEUTC$";
char* CustDate                    = "$WCDATE=%a, %d %B %Y$";
char* CustDateEmpty               = "$WCDATE=$";
char* CustDateInval               = "$WCDATE=%a, %c %B %Y$";
char* CustDateUTC                 = "$WCDATEUTC=%a, %d %B %Y$";
char* TimeNow                     = "$WCNOW$";
char* TimeNowUTC                  = "$WCNOWUTC$";
char* IsTagged                    = "$WCISTAGGED?Tagged:Not tagged$";
char* IsInGit                     = "$WCINGIT?versioned:not versioned$";
char* ModifiedFiles               = "$WCFILEMODS?Modified:Not modified$";
char* HasSubmodule                = "$WCSUBMODULE?Working tree has at least
 one submodule:Working tree has no submodules$";
char* SubmodulesUp2Date           = "$WCSUBMODULEUP2DATE?All submodules are
 up2date (checked out HEAD):At least one submodule is not up2date (checked
 HEAD differs)$";
char* SubmoduleHasModifications   = "$WCMODSINSUBMODULE?At least one
 submodule has uncommitted items:No submodule has uncommitted items$";
char* SubmoduleHasUnversioned     = "$WCUNVERINSUBMODULE?At least one
 submodule has unversioned files:No submodule with unversioned files$";
char* ModifiedAlsoInSubmodules    = "$WCMODSFULL?Modified items found
 (recursively):No modified items found (also not in submodules)$";
char* UnversionedAlsoInSubmodules = "$WCUNVERFULL?Unversioned items found
 (recursively):No unversioned items found (also not in submodules)$";

#if $WCMODSFULL?1:0$
#error Source is modified
```

```
#endif

// End of file
```

After running `GitWCRev.exe path\to\workingcopy testfile.tmpl testfile.txt`, the output file `testfile.txt` would looks like this:

```
// Test file for GitWCRev

char* Revision                    =
 "c16403bd41ba502935dee309fac137df0807f31e";
char* RevisionShort               = "c16403b";
char* Modified                    = "Modified";
char* Unversioned                 = "Unversioned items found";
char* Date                        = "2017/01/19 15:33:51";
char* DateUTC                     = "2017/01/19 14:33:51";
char* CustDate                    = "Thu, 19 January 2017";
char* CustDateEmpty               = "";
char* CustDateInval               = "Thu, 01/19/17 15:33:51 January 2017";
char* CustDateUTC                 = "Thu, 19 January 2017";
char* TimeNow                     = "2017/01/19 15:35:36";
char* TimeNowUTC                  = "2017/01/19 14:35:36";
char* IsTagged                    = "Not tagged";
char* IsInGit                     = "versioned";
char* ModifiedFiles               = "Not modified";
char* HasSubmodule                = "Working tree has at least one
 submodule";
char* SubmodulesUp2Date           = "At least one submodule is not up2date
 (checked HEAD differs)";
char* SubmoduleHasModifications   = "No submodule has uncommitted items";
char* SubmoduleHasUnversioned     = "At least one submodule has unversioned
 files";
char* ModifiedAlsoInSubmodules    = "Modified items found (recursively)";
char* UnversionedAlsoInSubmodules = "Unversioned items found
 (recursively)";

#if 1
#error Source is modified
#endif

// End of file
```

Tip

A file like this will be included in the build so you would expect it to be versioned. Be sure to version the template file, not the generated file, otherwise each time you regenerate the version file you need to commit the change, which in turn means the version file needs to be updated.

3.4. COM interface

If you need to access Git revision information from other programs, you can use the COM interface of GitWCRev. The object to create is `GitWCRev.object`, and the following methods are supported:

Method	Description
.GetWCInfo	This method checks and traverses the passed path gathering the status and revision information. Naturally you must call this before you can access the information using the remaining methods. The first parameter is the path. The

Method	Description
	second parameter needs to be true if you want to exclude submodules. Equivalent to the `-s` command line switch.
.Revision	The HEAD commit revision of the working tree. Equivalent to `$WCREV$`.
.Date	The commit date/time of the HEAD commit. Equivalent to `$WCDATE$`.
.Author	The author of the HEAD commit, that is, the last person to commit changes to the working tree.
.HasModifications	True if there are local modifications under the passed path
.HasUnversioned	True if there are unversioned items under the passed path
.IsGitItem	True if the item is versioned (false for directories except the working tree root).
.IsUnborn	True if the branch is not yet born.
.HasSubmodule	True if passed path contains submodules.
.HasSubmoduleModifications	True if a submodule has uncommitted changes.
.HasSubmoduleUnversioned	True if a submodule has unversioned items.
.IsSubmoduleUp2Date	True if all submodules are checked out at the in the parent repository specified version.

Table 3.4. COM/automation methods supported

The following example shows how the interface might be used.

```
// testCOM.js - javascript file
// test script for the GitWCRev COM/Automation-object

filesystem = new ActiveXObject("Scripting.FileSystemObject");

GitWCRevObject1 = new ActiveXObject("GitWCRev.object");
GitWCRevObject2 = new ActiveXObject("GitWCRev.object");
GitWCRevObject3 = new ActiveXObject("GitWCRev.object");
GitWCRevObject4 = new ActiveXObject("GitWCRev.object");
GitWCRevObject5 = new ActiveXObject("GitWCRev.object");

GitWCRevObject2_1 = new ActiveXObject("GitWCRev.object");
GitWCRevObject2_2 = new ActiveXObject("GitWCRev.object");
GitWCRevObject2_3 = new ActiveXObject("GitWCRev.object");
GitWCRevObject2_4 = new ActiveXObject("GitWCRev.object");
GitWCRevObject2_5 = new ActiveXObject("GitWCRev.object");

GitWCRevObject1.GetWCInfo(filesystem.GetAbsolutePathName("."), 0);
GitWCRevObject2.GetWCInfo(filesystem.GetAbsolutePathName(".."), 0);
GitWCRevObject3.GetWCInfo(filesystem.GetAbsolutePathName("GitWCRev.cpp"),
 0);
GitWCRevObject4.GetWCInfo(filesystem.GetAbsolutePathName("..\\.."), 0);

GitWCRevObject2_1.GetWCInfo(filesystem.GetAbsolutePathName("."), 1);
GitWCRevObject2_2.GetWCInfo(filesystem.GetAbsolutePathName(".."), 1);
GitWCRevObject2_3.GetWCInfo(filesystem.GetAbsolutePathName("GitWCRev.cpp"),
 1);
GitWCRevObject2_4.GetWCInfo(filesystem.GetAbsolutePathName("..\\.."), 1);

wcInfoString1 = "Revision = " + GitWCRevObject1.Revision +
                "\nDate = " + GitWCRevObject1.Date +
```

```
                "\nAuthor = " + GitWCRevObject1.Author +
                "\nHasMods = " + GitWCRevObject1.HasModifications +
                "\nHasUnversioned = " + GitWCRevObject1.HasUnversioned +
                "\nIsTagged = " + GitWCRevObject1.IsWcTagged +
                "\nIsGitItem = " + GitWCRevObject1.IsGitItem +
                "\nIsUnborn = " + GitWCRevObject1.IsUnborn +
                "\nHasSubmodule = " + GitWCRevObject1.HasSubmodule +
                "\nHasSubmoduleModifications = " +
 GitWCRevObject1.HasSubmoduleModifications +
                "\nHasSubmoduleUnversioned = " +
 GitWCRevObject1.HasSubmoduleUnversioned +
                "\nIsSubmoduleUp2Date = " +
 GitWCRevObject1.IsSubmoduleUp2Date;
wcInfoString2 = "Revision = " + GitWCRevObject2.Revision +
                "\nDate = " + GitWCRevObject2.Date +
                "\nAuthor = " + GitWCRevObject2.Author +
                "\nHasMods = " + GitWCRevObject2.HasModifications +
                "\nHasUnversioned = " + GitWCRevObject2.HasUnversioned +
                "\nIsTagged = " + GitWCRevObject2.IsWcTagged +
                "\nIsGitItem = " + GitWCRevObject2.IsGitItem +
                "\nIsUnborn = " + GitWCRevObject2.IsUnborn +
                "\nHasSubmodule = " + GitWCRevObject2.HasSubmodule +
                "\nHasSubmoduleModifications = " +
 GitWCRevObject2.HasSubmoduleModifications +
                "\nHasSubmoduleUnversioned = " +
 GitWCRevObject2.HasSubmoduleUnversioned +
                "\nIsSubmoduleUp2Date = " +
 GitWCRevObject2.IsSubmoduleUp2Date;
wcInfoString3 = "Revision = " + GitWCRevObject3.Revision +
                "\nDate = " + GitWCRevObject3.Date +
                "\nAuthor = " + GitWCRevObject3.Author +
                "\nHasMods = " + GitWCRevObject3.HasModifications +
                "\nHasUnversioned = " + GitWCRevObject3.HasUnversioned +
                "\nIsTagged = " + GitWCRevObject3.IsWcTagged +
                "\nIsGitItem = " + GitWCRevObject3.IsGitItem +
                "\nIsUnborn = " + GitWCRevObject3.IsUnborn +
                "\nHasSubmodule = " + GitWCRevObject3.HasSubmodule +
                "\nHasSubmoduleModifications = " +
 GitWCRevObject3.HasSubmoduleModifications +
                "\nHasSubmoduleUnversioned = " +
 GitWCRevObject3.HasSubmoduleUnversioned +
                "\nIsSubmoduleUp2Date = " +
 GitWCRevObject3.IsSubmoduleUp2Date;
wcInfoString4 = "Revision = " + GitWCRevObject4.Revision +
                "\nDate = " + GitWCRevObject4.Date +
                "\nAuthor = " + GitWCRevObject4.Author +
                "\nHasMods = " + GitWCRevObject4.HasModifications +
                "\nHasUnversioned = " + GitWCRevObject4.HasUnversioned +
                "\nIsTagged = " + GitWCRevObject4.IsWcTagged +
                "\nIsGitItem = " + GitWCRevObject4.IsGitItem +
                "\nIsUnborn = " + GitWCRevObject4.IsUnborn +
                "\nHasSubmodule = " + GitWCRevObject4.HasSubmodule +
                "\nHasSubmoduleModifications = " +
 GitWCRevObject4.HasSubmoduleModifications +
                "\nHasSubmoduleUnversioned = " +
 GitWCRevObject4.HasSubmoduleUnversioned +
                "\nIsSubmoduleUp2Date = " +
 GitWCRevObject4.IsSubmoduleUp2Date;
```

```
WScript.Echo(wcInfoString1 + "\n");
WScript.Echo(wcInfoString2 + "\n");
WScript.Echo(wcInfoString3 + "\n");
WScript.Echo(wcInfoString4 + "\n");

wcInfoString1 = "Revision = " + GitWCRevObject2_1.Revision +
                "\nDate = " + GitWCRevObject2_1.Date +
                "\nAuthor = " + GitWCRevObject2_1.Author +
                "\nHasMods = " + GitWCRevObject2_1.HasModifications +
                "\nHasUnversioned = " + GitWCRevObject2_1.HasUnversioned +
                "\nIsTagged = " + GitWCRevObject2_1.IsWcTagged +
                "\nIsGitItem = " + GitWCRevObject2_1.IsGitItem +
                "\nIsUnborn = " + GitWCRevObject2_1.IsUnborn +
                "\nHasSubmodule = " + GitWCRevObject2_1.HasSubmodule +
                "\nHasSubmoduleModifications = " +
 GitWCRevObject2_1.HasSubmoduleModifications +
                "\nHasSubmoduleUnversioned = " +
 GitWCRevObject2_1.HasSubmoduleUnversioned +
                "\nIsSubmoduleUp2Date = " +
 GitWCRevObject2_1.IsSubmoduleUp2Date;
wcInfoString2 = "Revision = " + GitWCRevObject2_2.Revision +
                "\nDate = " + GitWCRevObject2_2.Date +
                "\nAuthor = " + GitWCRevObject2_2.Author +
                "\nHasMods = " + GitWCRevObject2_2.HasModifications +
                "\nHasUnversioned = " + GitWCRevObject2_2.HasUnversioned +
                "\nIsTagged = " + GitWCRevObject2_2.IsWcTagged +
                "\nIsGitItem = " + GitWCRevObject2_2.IsGitItem +
                "\nIsUnborn = " + GitWCRevObject2_2.IsUnborn +
                "\nHasSubmodule = " + GitWCRevObject2_2.HasSubmodule +
                "\nHasSubmoduleModifications = " +
 GitWCRevObject2_2.HasSubmoduleModifications +
                "\nHasSubmoduleUnversioned = " +
 GitWCRevObject2_2.HasSubmoduleUnversioned +
                "\nIsSubmoduleUp2Date = " +
 GitWCRevObject2_2.IsSubmoduleUp2Date;
wcInfoString3 = "Revision = " + GitWCRevObject2_3.Revision +
                "\nDate = " + GitWCRevObject2_3.Date +
                "\nAuthor = " + GitWCRevObject2_3.Author +
                "\nHasMods = " + GitWCRevObject2_3.HasModifications +
                "\nHasUnversioned = " + GitWCRevObject2_3.HasUnversioned +
                "\nIsTagged = " + GitWCRevObject2_3.IsWcTagged +
                "\nIsGitItem = " + GitWCRevObject2_3.IsGitItem +
                "\nIsUnborn = " + GitWCRevObject2_3.IsUnborn +
                "\nHasSubmodule = " + GitWCRevObject2_3.HasSubmodule +
                "\nHasSubmoduleModifications = " +
 GitWCRevObject2_3.HasSubmoduleModifications +
                "\nHasSubmoduleUnversioned = " +
 GitWCRevObject2_3.HasSubmoduleUnversioned +
                "\nIsSubmoduleUp2Date = " +
 GitWCRevObject2_3.IsSubmoduleUp2Date;
wcInfoString4 = "Revision = " + GitWCRevObject2_4.Revision +
                "\nDate = " + GitWCRevObject2_4.Date +
                "\nAuthor = " + GitWCRevObject2_4.Author +
                "\nHasMods = " + GitWCRevObject2_4.HasModifications +
                "\nHasUnversioned = " + GitWCRevObject2_4.HasUnversioned +
                "\nIsTagged = " + GitWCRevObject2_4.IsWcTagged +
                "\nIsGitItem = " + GitWCRevObject2_4.IsGitItem +
```

```
                "\nIsUnborn = " + GitWCRevObject2_4.IsUnborn +
                "\nHasSubmodule = " + GitWCRevObject2_4.HasSubmodule +
                "\nHasSubmoduleModifications = " +
 GitWCRevObject2_4.HasSubmoduleModifications +
                "\nHasSubmoduleUnversioned = " +
 GitWCRevObject2_4.HasSubmoduleUnversioned +
                "\nIsSubmoduleUp2Date = " +
 GitWCRevObject2_4.IsSubmoduleUp2Date;

WScript.Echo(wcInfoString1 + "\n");
WScript.Echo(wcInfoString2 + "\n");
WScript.Echo(wcInfoString3 + "\n");
WScript.Echo(wcInfoString4 + "\n");
```

The following listing is an example on how to use the GitWCRev COM object from C#:

```
using LibGitWCRev;
GitWCRev sub = new GitWCRev();
sub.GetWCInfo("C:\\PathToMyFile\\MyFile.cc", false);
if (sub.IsGitItem == true)
{
    MessageBox.Show("versioned");
}
else
{
    MessageBox.Show("not versioned");
}
```

Appendix A. Frequently Asked Questions (FAQ)

Because TortoiseGit is being developed all the time it is sometimes hard to keep the documentation completely up to date. *online FAQ* [https://tortoisegit.org/faq] which contains a selection of the questions we are asked the most on the TortoiseGit mailing lists `<tortoisegit-dev@googlegroups.com>` and `<tortoisegit-users@googlegroups.com>`.

We also maintain a project *Issue Tracker* [https://tortoisegit.org/issues] which tells you about some of the things we have on our To-Do list, and bugs which have already been fixed. If you think you have found a bug, or want to request a new feature, check here first to see if someone else got there before you.

If you have a question which is not answered anywhere else, the best place to ask it is on one of the mailing lists. `<tortoisegit-users@googlegroups.com>` is the one to use if you have questions about using TortoiseGit. If you want to help out with the development of TortoiseGit, then you should take part in discussions on `<tortoisegit-dev@googlegroups.com>`.

Appendix B. IBugTraqProvider interface

To get a tighter integration with issue trackers than by simply using the `bugtraq.` config keys, TortoiseGit can make use of COM plugins. With such plugins it is possible to fetch information directly from the issue tracker, interact with the user and provide information back to TortoiseGit about open issues, verify log messages entered by the user and even run actions after a successful commit to e.g, close an issue.

We can't provide information and tutorials on how you have to implement a COM object in your preferred programming language, but we have example plugins in C++/ATL and C# in our repository in the `contrib/issue-tracker-plugins` folder. In that folder you can also find the required include files you need to build your plugin. (Section 3, "TortoiseGit is free!" explains how to access the repository.)

Important

You should provide both a 32-bit and 64-bit version of your plugin. Because the x64-Version of TortoiseGit cannot use a 32-bit plugin and vice-versa.

B.1. Naming conventions

If you release an issue tracker plugin for Tortoise*-clients, please do *not* name it *Tortoise<Something>*. We'd like to reserve the *Tortoise* prefix for a version control client integrated into the windows shell. For example: TortoiseCVS, TortoiseSVN, TortoiseHg, TortoiseGit and TortoiseBzr are all version control clients.

Please name your plugin for a Tortoise client *Turtle<Something>*, where *<Something>* refers to the issue tracker that you are connecting to. Alternatively choose a name that sounds like *Turtle* but has a different first letter. Nice examples are:

- TurtleMine - An issue tracker plugin for Redmine
- VurtleOne - An issue tracker plugin for VersionOne

B.2. The IBugTraqProvider interface

TortoiseGit 1.2.1 and later can use plugins which implement the IBugTraqProvider interface. The interface provides a few methods which plugins can use to interact with the issue tracker.

```
HRESULT ValidateParameters (
  // Parent window for any UI that needs to be
  // displayed during validation.
  [in] HWND hParentWnd,

  // The parameter string that needs to be validated.
  [in] BSTR parameters,

  // Is the string valid?
  [out, retval] VARIANT_BOOL *valid
);
```

This method is called from the settings dialog where the user can add and configure the plugin. The `parameters` string can be used by a plugin to get additional required information, e.g., the URL to the issue tracker, login information, etc. The plugin should verify the `parameters` string and show an error dialog if the string is not valid. The `hParentWnd` parameter should be used for any dialog the plugin shows as the parent window. The plugin must return TRUE if the validation of the `parameters` string is successful. If the plugin returns FALSE, the settings dialog won't allow the user to add the plugin to a working copy path.

```
HRESULT GetLinkText (
  // Parent window for any (error) UI that needs to be displayed.
  [in] HWND hParentWnd,

  // The parameter string, just in case you need to talk to your
  // web service (e.g.) to find out what the correct text is.
  [in] BSTR parameters,

  // What text do you want to display?
  // Use the current thread locale.
  [out, retval] BSTR *linkText
);
```

The plugin can provide a string here which is used in the TortoiseGit commit dialog for the button which invokes the plugin, e.g., "Choose issue" or "Select ticket". Make sure the string is not too long, otherwise it might not fit into the button. If the method returns an error (e.g., `E_NOTIMPL`), a default text is used for the button.

```
HRESULT GetCommitMessage (
  // Parent window for your provider's UI.
  [in] HWND hParentWnd,

  // Parameters for your provider.
  [in] BSTR parameters,
  [in] BSTR commonRoot,
  [in] SAFEARRAY(BSTR) pathList,

  // The text already present in the commit message.
  // Your provider should include this text in the new message,
  // where appropriate.
  [in] BSTR originalMessage,

  // The new text for the commit message.
  // This replaces the original message.
  [out, retval] BSTR *newMessage
);
```

This is the main method of the plugin. This method is called from the TortoiseGit commit dialog when the user clicks on the plugin button.

The `parameters` string is the string the user has to enter in the settings dialog when he configures the plugin. Usually a plugin would use this to find the URL of the issue tracker and/or login information or more.

The `commonRoot` string contains the parent path of all items selected to bring up the commit dialog. Note that this is *not* the root path of all items which the user has selected in the commit dialog. For the branch/tag dialog, this is the path which is to be copied.

The `pathList` parameter contains an array of paths (as strings) which the user has selected for the commit.

The `originalMessage` parameter contains the text entered in the log message box in the commit dialog. If the user has not yet entered any text, this string will be empty.

The `newMessage` return string is copied into the log message edit box in the commit dialog, replacing whatever is already there. If a plugin does not modify the `originalMessage` string, it must return the same string again here, otherwise any text the user has entered will be lost.

B.3. The IBugTraqProvider2 interface

In TortoiseSVN 1.6 a new interface was added which provides more functionality for plugins (also available in TortoiseGit since 1.2.1). This IBugTraqProvider2 interface inherits from IBugTraqProvider.

```
HRESULT GetCommitMessage2 (
  // Parent window for your provider's UI.
  [in] HWND hParentWnd,

  // Parameters for your provider.
  [in] BSTR parameters,
  // The common URL of the commit
  [in] BSTR commonURL,
  [in] BSTR commonRoot,
  [in] SAFEARRAY(BSTR) pathList,

  // The text already present in the commit message.
  // Your provider should include this text in the new message,
  // where appropriate.
  [in] BSTR originalMessage,

  // You can assign custom revision properties to a commit
  // by setting the next two params.
  // note: Both safearrays must be of the same length.
  //       For every property name there must be a property value!

  // The content of the bugID field (if shown)
  [in] BSTR bugID,

  // Modified content of the bugID field
  [out] BSTR * bugIDOut,

  // The list of revision property names.
  [out] SAFEARRAY(BSTR) * revPropNames,

  // The list of revision property values.
  [out] SAFEARRAY(BSTR) * revPropValues,

  // The new text for the commit message.
  // This replaces the original message
  [out, retval] BSTR * newMessage
);
```

This method is called from the TortoiseGit commit dialog when the user clicks on the plugin button. This method is called instead of `GetCommitMessage()`. Please refer to the documentation for `GetCommitMessage` for the parameters that are also used there.

The parameter `commonURL` is the parent URL of all items selected to bring up the commit dialog. This is basically the URL of the `commonRoot` path.

The parameter `bugID` contains the content of the bug-ID field (if it is shown, configured with the property `bugtraq.message`).

The return parameter `bugIDOut` is used to fill the bug-ID field when the method returns.

The `revPropNames` and `revPropValues` are only honored by TortoiseSVN and are ignored by TortoiseGit. If no revision properties are to be set, the plugin must return empty arrays.

```
HRESULT CheckCommit (
  [in] HWND hParentWnd,
  [in] BSTR parameters,
  [in] BSTR commonURL,
```

```
  [in] BSTR commonRoot,
  [in] SAFEARRAY(BSTR) pathList,
  [in] BSTR commitMessage,
  [out, retval] BSTR * errorMessage
);
```

This method is called right before the commit dialog is closed and the commit begins. A plugin can use this method to validate the selected files/folders for the commit and/or the commit message entered by the user. The parameters are the same as for `GetCommitMessage2()`, with the difference that `commonURL` is now the common URL of all *checked* items, and `commonRoot` the root path of all checked items.

For the branch/tag dialog, the `commonURL` is the source URL of the copy, and `commonRoot` is set to the target URL of the copy.

The return parameter `errorMessage` must either contain an error message which TortoiseGit shows to the user or be empty for the commit to start. If an error message is returned, TortoiseGit shows the error string in a dialog and keeps the commit dialog open so the user can correct whatever is wrong. A plugin should therefore return an error string which informs the user *what* is wrong and how to correct it.

```
HRESULT  OnCommitFinished (
  // Parent window for any (error) UI that needs to be displayed.
  [in] HWND hParentWnd,

  // The common root of all paths that got committed.
  [in] BSTR commonRoot,

  // All the paths that got committed.
  [in] SAFEARRAY(BSTR) pathList,


  // The text already present in the commit message.
  [in] BSTR logMessage,

  // The revision of the commit.
  [in] ULONG revision,


  // An error to show to the user if this function
  // returns something else than S_OK
  [out, retval] BSTR * error
);
```

This method is called after a successful commit. A plugin can use this method to e.g., close the selected issue or add information about the commit to the issue. The parameters are the same as for `GetCommitMessage2`.

```
HRESULT HasOptions(
  // Whether the provider provides options
  [out, retval] VARIANT_BOOL *ret
);
```

This method is called from the settings dialog where the user can configure the plugins. If a plugin provides its own configuration dialog with `ShowOptionsDialog`, it must return TRUE here, otherwise it must return FALSE.

```
HRESULT ShowOptionsDialog(
  // Parent window for the options dialog
  [in] HWND hParentWnd,
```

```
  // Parameters for your provider.
  [in] BSTR parameters,

  // The parameters string
  [out, retval] BSTR * newparameters
);
```

This method is called from the settings dialog when the user clicks on the "Options" button that is shown if `HasOptions` returns TRUE. A plugin can show an options dialog to make it easier for the user to configure the plugin.

The `parameters` string contains the plugin parameters string that is already set/entered.

The `newparameters` return parameter must contain the parameters string which the plugin constructed from the info it gathered in its options dialog. That `paramameters` string is passed to all other IBugTraqProvider and IBugTraqProvider2 methods.

Appendix C. Useful Tips For Administrators

This appendix contains solutions to problems/questions you might have when you are responsible for deploying TortoiseGit to multiple client computers.

C.1. Deploy TortoiseGit via group policies

The TortoiseGit installer comes as an MSI file, which means you should have no problems adding that MSI file to the group policies of your domain controller.

A good walk-through on how to do that can be found from Microsoft: *https://learn.microsoft.com/en-us/troubleshoot/windows-server/group-policy/use-group-policy-to-install-software* [https://learn.microsoft.com/en-us/troubleshoot/windows-server/group-policy/use-group-policy-to-install-software].

Versions 0.3.0 and later of TortoiseGit must be installed under *Computer Configuration* and not under *User Configuration*. This is because those versions need the new CRT and MFC DLLs, which can only be deployed *per computer* and not *per user*. If you really must install TortoiseGit on a per user basis, then you must first install the MFC and CRT package version 11 from Microsoft on each computer you want to install TortoiseGit as per user.

You can provide a default setting for the SSH client in `HKLM\TortoiseGit\SSH`.

TortoiseGit automatically finds `git.exe` if a normal msysGit/Git for Windows installation is on the computer or `git.exe` is on the `PATH` (and is runnable in a normal `cmd.exe` session - you might need to also put the `[MSYSGIT INSTALLDIR]\mingw\bin` on the `PATH` if you use the msysGit development package).

For completely disabling automatic update checking see `VersionCheck` in Section 2.36.10, “Advanced Settings”.

C.2. Redirect the upgrade check

TortoiseGit checks if there's a new version available every week (or daily in a preview release). If there is a newer version available, a dialog shows up informing the user about that and allows to download/install a new version.

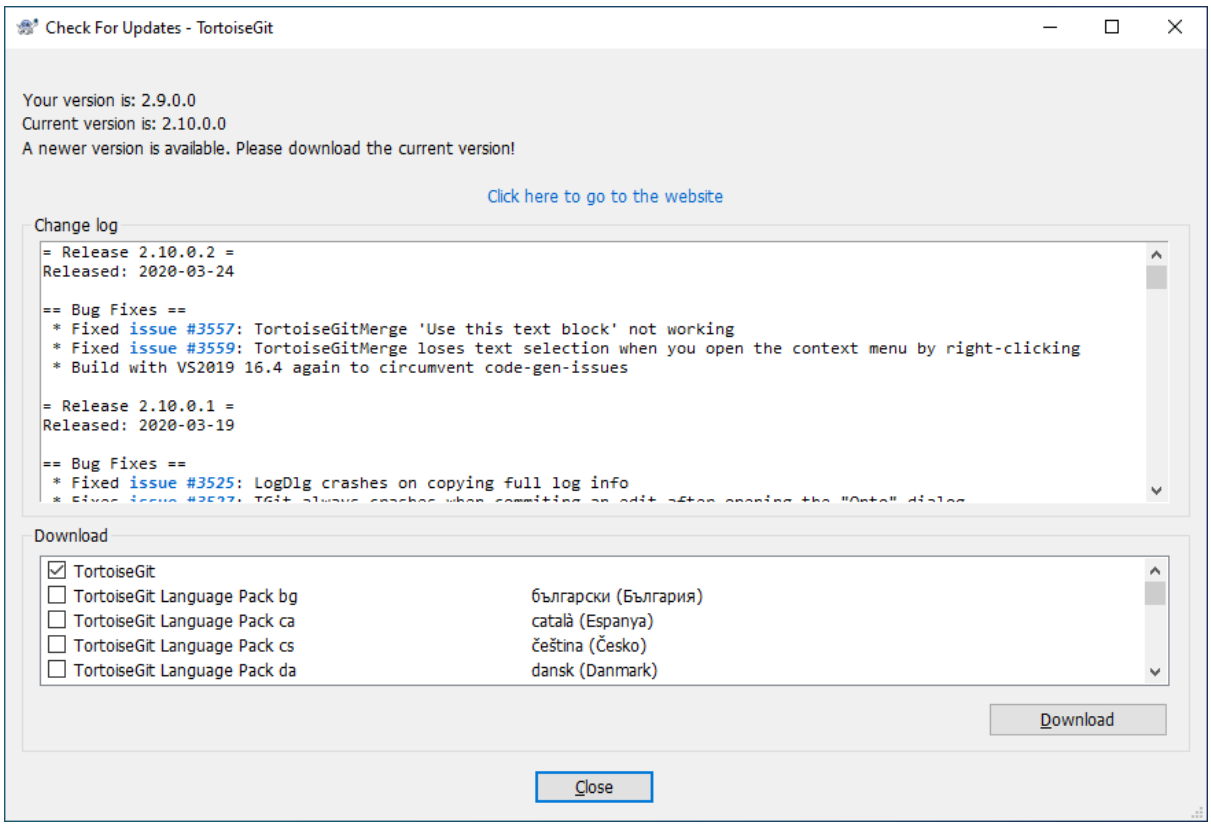


Figure C.1. The upgrade dialog

If you're responsible for a lot of users in your domain, you might want your users to use only versions you have approved and not have them install always the latest version (or to save bandwidth or want to add some further notes for installation). You probably don't want that upgrade dialog to show up so your users don't go and upgrade immediately (to disable update checking at all (e.g. because you use group policies to deploy TortoiseGit, see Section C.1, “Deploy TortoiseGit via group policies” and/or `VersionCheck` in Section 2.36.10, “Advanced Settings”).

TortoiseGit allow you to redirect that upgrade check to your intranet server. You can set the registry key `HKCU\Software\TortoiseGit\UpdateCheckURL` OR `HKLM\Software\TortoiseGit\UpdateCheckURL` (string value, `HKCU` overrides `HKLM`) to an URL pointing to a text file in your intranet (default is https://versioncheck.tortoisegit.org/version.txt). When the default `version.txt` file is used, it is checked by verifying a digital signature (https://versioncheck.tortoisegit.org/version.txt.rsa.asc) that it has not been altered (since TortoiseGit 1.8.5). The check for the digital signature of the `version.txt` file is omitted if the location is overridden in registry. That text file must have the following format:

```
[TortoiseGit]
version=X.X.X.X
infotext=A new version of TortoiseGit is available for you to download!
infotexturl=http://192.168.2.1/downloads/TortoiseGit/info.htm
changelogurl=http://192.168.2.1/downloads/TortoiseGit/
TortoiseGit-1.4.1.6000-changelog.txt
baseurl=http://192.168.2.1/downloads/TortoiseGit/
langs="1029;cs"
langs="1031;de"
```

The `version` line in that file is the version string. You must make sure that it matches the exact version string of the TortoiseGit installation package. The `infotext` line is a custom text, shown in the upgrade dialog. You can write there whatever you want (can also be left empty). Just note that the space in the upgrade dialog is limited. Too long messages will get truncated! The `infotexturl` line is the URL which is opened when when the user

clicks on the (custom) message label in the upgrade dialog. The URL is opened with the default web browser, so if you specify a web page, that page is opened and shown to the user. The `changelogurl` line contains the URL to the changelog or release notes which are displayed in the upgrade dialog (if empty it defaults to https://versioncheck.tortoisegit.org/changelog.txt, you can use `%1!u!`, `%2!u!` and `%3!u!` for MAJOR, MINOR and MICRO version numbers of the running TortoiseGit version; `%4!s!` for Windows platform, `%5!s!` for Windows version, and `%6!s!` for service pack version), The `baseurl` line is used to override the base path to the installation packages (if empty it defaults to `https://updater.download.tortoisegit.org/tgit/X.X.X.X/`). The filenames are generated as follows: `TortoiseGit-(version)-(32|64)bit.msi` for the main installer (if not overridden by `mainfilename=TortoiseGit-%1!s!-%2!s!bit.msi`) and `TortoiseGit-LanguagePack-(version)-(32|64)bit-(cs|de|...).msi` for the language packs (if not overridden by `languagepackfilename=TortoiseGit-LanguagePack-%1!s!-%2!s!bit-%3!s!.msi`; `%4!d!` is the four digit country code). Using `langs` lines, one can advertise language packs (Syntax of one line: Four digit country code;ISO Country code). Using a `issuesurl` line, it is possible to control the URL to which the issues are linked to (default is `https://tortoisegit.org/issue/%BUGID%`; can also be empty to disable linking). There are also two boolean options `directdownload` and `changelog` to hide the change log and to disable direct downloads (both default to true).

Clicking on Download downloads the selected files as well as their digital signature files (`filename.asc`) to `FOLDERID_Downloads` [https://docs.microsoft.com/windows/win32/shell/knownfolderid]. After downloading the digital signature is verified - the file is only kept if the file is digitally signed and could be verified correctly.

If you want to distribute your own modified TortoiseGit packages in your network, you have to put your own GPG key into TortoiseGit and sign the `.msi`-files with this key or deactivate the signature verification completely.

C.3. Disable context menu entries

TortoiseGit allows you to disable (actually, hide) context menu entries. Since this is a feature which should not be used lightly but only if there is a compelling reason, there is no GUI for this and it has to be done directly in the registry. This can be used to disable certain commands for users who should not use them. But please note that only the context menu entries in the *explorer* are hidden, and the commands are still available through other means, e.g. the command line or even other dialogs in TortoiseGit itself!

The registry keys which hold the information on which context menus to show are `HKEY_CURRENT_USER\Software\TortoiseGit\ContextMenuEntriesMaskLow` and `HKEY_CURRENT_USER\Software\TortoiseGit\ContextMenuEntriesMaskHigh`.

Each of these registry entries is a `DWORD` value, with each bit corresponding to a specific menu entry. A set bit means the corresponding menu entry is deactivated.

Value	Menu entry
0x0000000000000002	Sync
0x0000000000000004	Commit
0x0000000000000008	Add
0x0000000000000010	Revert
0x0000000000000020	Cleanup
0x0000000000000040	Resolve
0x0000000000000080	Switch/Checkout
0x0000000000000100	Sendmail
0x0000000000000200	Export
0x0000000000000400	Create Repository here
0x0000000000000800	Branch/Tag
0x0000000000001000	Merge
0x0000000000002000	Delete

Value	Menu entry
0x0000000000004000	Rename
0x0000000000008000	Submodule Update
0x0000000000010000	Diff
0x0000000000020000	Show Log
0x0000000000040000	Edit Conflicts
0x0000000000080000	Reference Browser
0x0000000000100000	Check for modifications
0x0000000000200000	Ignore
0x0000000000400000	RefLog
0x0000000000800000	Blame
0x0000000001000000	Repository Browser
0x0000000002000000	Apply Patch
0x0000000004000000	Delete (keep local)
0x0000000008000000	SVN Rebase
0x0000000010000000	SVN DCommit
0x0000000040000000	SVN Ignore
0x0000000100000000	Log of Submodule folder
0x0000000200000000	Rev Diff
0x0000000800000000	Pull
0x0000001000000000	Push
0x0000002000000000	Clone
0x0000004000000000	Tag
0x0000008000000000	Format Patch
0x0000010000000000	Import Patch
0x0000040000000000	Fetch
0x0000080000000000	Rebase
0x0000100000000000	Stash Save
0x0000200000000000	Stash Apply
0x0000400000000000	Stash List
0x0000800000000000	Submodule Add
0x0001000000000000	Submodule Sync
0x0002000000000000	Stash Pop
0x0004000000000000	Diff two files
0x0008000000000000	Bisect
0x0080000000000000	SVN Fetch
0x0100000000000000	Revision graph
0x0200000000000000	Daemon
0x2000000000000000	Settings
0x4000000000000000	Help

Value	Menu entry
0x8000000000000000	About

Table C.1. Menu entries and their values

Example: to disable the `Sendmail`, the `Rebase` and the `Settings` menu entries, add the values assigned to the entries like this:

```
  0x0000000000000100
+ 0x0000080000000000
+ 0x2000000000000000
= 0x2000080000000100
```

The lower `DWORD` value (`0x00000100`) must then be stored in `HKEY_CURRENT_USER\Software\TortoiseGit\ContextMenuEntriesMaskLow`, the higher `DWORD` value (`0x20000800`) in `HKEY_CURRENT_USER\Software\TortoiseGit\ContextMenuEntriesMaskHigh`.

To enable the menu entries again, simply delete the two registry keys.

Appendix D. Automating TortoiseGit

Since all commands for TortoiseGit are controlled through command line parameters, you can automate it with batch scripts or start specific commands and dialogs from other programs (e.g. your favorite text editor).

> **Important**
>
> Remember that TortoiseGit is a GUI client, and this automation guide shows you how to make the TortoiseGit dialogs appear to collect user input. If you want to write a script which requires no input, you should use the official Git command line client instead.

D.1. TortoiseGit Commands

The TortoiseGit GUI program is called `TortoiseGitProc.exe`. All commands are specified with the parameter `/command:abcd` where `abcd` is the required command name. Most of these commands need at least one path argument, which is given with `/path:"some\path"`. In the following table the command refers to the `/command:abcd` parameter and the path refers to the `/path:"some\path"` parameter.

Since some of the commands can take a list of target paths (e.g. committing several specific files) the `/path` parameter can take several paths, separated by a `*` character.

TortoiseGit uses temporary files to pass multiple arguments between the shell extension and the main program. From TortoiseGit 1.5.0 on and later, `/notempfile` parameter is obsolete and there is no need to add it anymore.

The progress dialog which is used for commits, updates and many more `git.exe` commands usually stays open after the command has finished until the user presses the OK button. This can be changed in the settings dialog. You may use `/closeonend` parameter to override the this setting from your batch file.

To close the (`git.exe`) progress dialog at the end of a command automatically without using the permanent setting you can pass the `/closeonend` parameter.

- `/closeonend:0` Close manually
- `/closeonend:1` Auto-close if no further options are available
- `/closeonend:2` Auto-close if no errors

The table below lists all the commands which can be accessed using the TortoiseGitProc.exe command line. As described above, these should be used in the form `/command:abcd`. In the table, the `/command` prefix is omitted to save space.

Command	Description
:about	Shows the about dialog. This is also shown if no command is given.
:bisect	Allows to control the bisect logic of TortoiseGit. Use the `/start` parameter to start a bisect you can specify `/good:REF` and `/bad:REF` here). When bisect is active, you can use `/good`, `/bad`, `/skip` and `/reset` to control the bisect process.
:fetch	Opens the fetch dialog. Use the `/remote` parameter to control the remote which should be pre-selected.
:firststart	Shows the first start wizard.
:log	Opens the log dialog. The `/path` specifies the file or folder for which the log should be shown. Additional options can be set: `/rev:"SHA1"` highlights and automatically scrolls to the specified revision, `/endrev:"SHA1/branch"`, shows the log of the specified revision, `/startrev:"SHA1/branch"` (only in combination with `endrev`), shows the log of the revision range `startrev..endrev`, `/range:"gitrevision"`, shows the log of the entered `gitrevision` (e.g. "`branch1...branch2`"), `/limit:"N SCALE"`, SCALE could be "Commit", "Year", "Month", "Week"; it shows last N commit(s), last N year(s), last

Command	Description
	N month(s), last N week(s). Use `/limit:0` to disable any default limit. `/findstring:"filterstring"` fills in the filter text, `/findtext` forces the filter to use text, not regex, or `/findregex` forces the filter to use regex, not simple text search, and `/findtype:X` with X being a number between 0 and 127. The numbers are the sum of the following options: • `/findtype:0` filter by everything • `/findtype:1` filter by messages • `/findtype:2` filter by path • `/findtype:4` filter by authors • `/findtype:8` filter by revisions • `/findtype:16` not used • `/findtype:32` filter by bug ID • `/findtype:64` filter by subject If `/outfile:path\to\file` is specified, the selected revision is written to that file when the log dialog is closed.
:clone	Opens the clone dialog. The `/url` specifies the URL to clone from. The `/path` specifies the target directory to clone to. If `/exactpath` is not specified, the repository name (without trailing .git) will be appended to target directory. This is the default behavior. If `/exactpath` is specified, the exact `/path` is considered the target directory, without appending repository name to it.
:commit	Opens the commit dialog. The `/path` specifies the target directory or the list of files to commit. You can also specify the `/logmsg` switch to pass a predefined log message to the commit dialog. Or, if you don't want to pass the log message on the command line, use `/logmsgfile:path`, where `path` points to a file containing the log message. To pre-fill the bug ID box (in case you've set up integration with bug trackers properly), you can use the `/bugid:"the bug id here"` to do that.
:add	Adds the files in `/path` to version control.
:revert	Reverts local modifications of a working tree. The `/path` tells which items to revert.
:cleanup	Cleans up the working tree in `/path`.
:resolve	Marks a conflicted file specified in `/path` as resolved. If `/noquestion` is given, then resolving is done without asking the user first if it really should be done.
:repocreate	Creates a repository in `/path`
:switch	Opens the switch dialog. The `/path` specifies the target directory.
:export	Exports a revision of the repository in `/path` to a zip file.
:merge	Opens the merge dialog. The `/path` specifies the target directory. The `/abort` opens abort merge dialog.
:settings	Opens the settings dialog.
:remove	Removes the file(s) in `/path` from version control.
:rename	Renames the file in `/path`. The new name for the file is asked with a dialog.
:diff	Starts the external diff program specified in the TortoiseGit settings. The `/path` specifies the first file. If the option `/path2` is set, then the diff program is start-

Command	Description
	ed with those two files. If `/path2` is omitted, then the diff is done between the file in `/path` and its BASE. To explicitly set the revision use `/startrev:xxx` (BASE revision) and `/endrev:xxx`. Add `/unified` to get a unified diff. Add `/line:NN` to get scroll to the mentioned line.
:showcompare	Depending on revisions to compare and the path, this either shows a unified diff (if the option `unified` is set), a dialog with a list of files that have changed (filtered by a possibly entered sub path) or if the path point to a file starts the diff viewer for those the file in the different revisions. Use `/revision1:xxx` and `/revision2:xxx` to specify the revisions to compare, whereas `/revision1:xxx` indicates the base revision to compare with.
:conflicteditor	Starts the conflict editor specified in the TortoiseGit settings with the correct files for the conflicted file in `/path`.
:help	Opens the help file.
:repostatus	Opens the check-for-modifications dialog. The `/path` specifies the working tree directory.
:repobrowser	Starts the repository browser dialog, pointing to the working tree given in `/path`. An additional option `/rev:xxx` can be used to specify the revision which the repository browser should show. If the `/rev:xxx` is omitted, it defaults to HEAD.
:ignore	Adds all targets in `/path` to the ignore list, i.e. adds file(s) to the `.gitignore` file.
:blame	Opens TortoiseGitBlame for the file specified in `/path`. If the option `/endrev` is set TortoiseGitBlame ends at that revision. If the option `/line:nnn` is set, TortoiseGitBlame will open with the specified line number showing.
:cat	Saves a file from an URL or working tree path given in `/path` to the location given in `/savepath:path`. The revision is given in `/revision:xxx`. This can be used to get a file with a specific revision.
:pull	Opens the pull dialog in the working tree located in `/path`.
:push	Opens the push dialog in the working tree located in `/path`.
:rebase	Opens the rebase dialog for the working tree located in `/path`. To select the upstream branch/commit use the `/upstream` option. For activating the force option use `/force`.
:stashsave	Opens the stash save dialog for the working tree located in `/path`. A pre-filled message can be achieved by using the `/msg` parameter.
:stashapply	Applies to latest stash to the working tree located in `/path`.
:stashpop	Applies to latest stash to the working tree located in `/path` and drops the latest stash entry.
:subadd	Opens the submodule add dialog. `/path`.
:subupdate	Opens the submodule update dialog for and filters the submodules regarding the folder `/path`.
:subsync	Syncs the submodule information for the working tree located in `/path`.
:reflog	Opens the RefLog dialog for the repository located in `/path`.
:refbrowse	Opens the browse references dialog for the repository located in `/path`.
:updatecheck	`/visible`: Shows the dialog even if no newer TortoiseGit version is available. `/force`: Shows file list for download even if the latest TortoiseGit has been installed.

Command	Description
:revisiongraph	Shows the revision graph for the repository given in `/path`. To create an image file of the revision graph for a specific path, but without showing the graph window, pass `/output:path` with the path to the output file. The output file must have an extension that the revision graph can actually export to. These are: `.svg`, `.wmf`, `.gv`, `.png`, `.jpg`, `.bmp` and `.gif`.
:daemon	Launches the Git Daemon for the repository given in `/path`.
:pgpfp	Prints the TortoiseGit Release Signing Key fingerprint. If you trust the current TortoiseGit installation, this can be used as a trust anchor to future releases.
:tag	Opens the Create Tag dialog. The `/path` specifies the repository folder. Additional options can be set: `/rev:"ref"` tags on the specified ref/commit, `/name:"tag_name"` fills the Tag name in Create Tag dialog.
:branch	Opens the Create Branch dialog. The `/path` specifies the repository folder. Additional options can be set: `/rev:"ref"` tags on the specified ref/commit, `/name:"tag_name"` fills the Tag name in Create Tag dialog.
:lfslocks	Opens the Show Locks dialog. More info in LFS Locking.
:worktreelist	Opens the Worktree List dialog.

Table D.1. List of available commands and options

Examples (which should be entered on one line):

```
TortoiseGitProc.exe /command:commit
                 /path:"d:\git_wc\file1.txt*c:\git_wc\file2.txt"
                 /logmsg:"test log message" /closeonend:2

TortoiseGitProc.exe /command:log /path:"c:\git_wc\file1.txt"
                 /startrev:master~100 /endrev:master
```

Tip

When calling TortoiseGit from within the MSYS environment, you can also use more *nix style command line parameters:

```
TortoiseGitProc.exe -command commit
                    -path "d:\git_wc\file1.txt*c:\git_wc
\file2.txt"
                    -logmsg "test log message" -closeonend 2
```

D.2. TortoiseGitIDiff Commands

The image diff tool has a few command line options which you can use to control how the tool is started. The program is called `TortoiseGitIDiff.exe`.

The table below lists all the options which can be passed to the image diff tool on the command line.

Option	Description
:left	Path to the file shown on the left.
:lefttitle	A title string. This string is used in the image view title instead of the full path to the image file.

Option	Description
:right	Path to the file shown on the right.
:righttitle	A title string. This string is used in the image view title instead of the full path to the image file.
:overlay	If specified, the image diff tool switches to the overlay mode (alpha blend).
:fit	If specified, the image diff tool fits both images together.
:showinfo	Shows the image info box.

Table D.2. List of available options

Example (which should be entered on one line):

```
TortoiseGitIDiff.exe /left:"c:\images\img1.jpg" /lefttitle:"image 1"
                     /right:"c:\images\img2.jpg" /righttitle:"image 2"
                     /fit /overlay
```

Appendix E. Implementation Details

This appendix contains a more detailed discussion of the implementation of some of TortoiseGit's features.

E.1. Icon Overlays

Every file has a Git status value as reported by the Git library. In the command line client, these are represented by single letter codes, but in TortoiseGit they are shown graphically using the icon overlays. Because the number of overlays is very limited, each overlay may represent one of several status values.

The *Conflicted* overlay is used to represent the `conflicted` state, where a merge resulted in conflicts between the changes of the current and changes from another branch.

The *Modified* overlay represents the `modified` state, where you have made local modifications to your working tree.

The *Deleted* overlay represents the `deleted` state, where an item is scheduled for deletion, or the `missing` state, where an item is not present but still in the Git index. Naturally an item which is missing cannot have an overlay itself, but the parent folder can be marked if one of its child items is missing.

The *Added* overlay is simply used to represent the `added` status when an item has been added to version control.

The *In Git* overlay is used to represent an item which is in the `normal` state.

The *assume-valid* (*Needs Lock* in TortoiseSVN) overlay is used to indicate if a file has the `assume-valid` flag set.

The *skip-worktree* (*Locked* in TortoiseSVN) overlay is used when to indicate if a file has the `skip-worktree` flag set.

The *Ignored* overlay is used to represent an item which is in the `ignored` state, either due to a global ignore pattern, or due to a `.gitignore` file in one of the parent folders. This overlay is optional.

The *Unversioned* overlay is used to represent an item which is in the `unversioned` state. This is an item in a versioned folder, but which is not under version control itself. This overlay is optional.

If an item has Git status `none` (the item is not within a working tree) then no overlay is shown. If you have chosen to disable the *Ignored* and *Unversioned* overlays then no overlay will be shown for those files either.

An item can only have one Git status value. For example a file could be locally modified and it could be marked for deletion at the same time. Git returns a single status value - in this case `deleted`. Those priorities are defined within Git and TortoiseGit itself.

When TortoiseGit displays the status recursively (the default setting), each folder displays an overlay reflecting its own status and the status of all its children. In order to display a single *summary* overlay, we use the priority order shown above to determine which overlay to use, with the *Conflicted* overlay taking highest priority.

In fact, you may find that not all of these icons are used on your system. This is because the number of overlays allowed by Windows is limited to 15. Windows uses 4 of those, and the remaining 11 can be used by other applications. If there are not enough overlay slots available, TortoiseGit tries to be a *Good Citizen (TM)* and limits its use of overlays to give other apps a chance.

If you have problems with overlays, please see the online *FAQ* [https://tortoisegit.org/support/faq/#ovlnotshowing].

Since there are Tortoise clients available for other version control systems, the TortoiseSVN developers created a shared component which is responsible for showing the overlay icons. The technical details are not important here, all you need to know is that this shared component allows all Tortoise clients to use the same overlays and therefore the limit of 11 available slots isn't used up by installing more than one Tortoise client. Of course there's one small drawback: all Tortoise clients use the same overlay icons, so you can't figure out by the overlay icons what version control system a working copy is using.

- *Normal*, *Modified* and *Conflicted* are always loaded and visible.
- *Deleted* is loaded if possible, but falls back to *Modified* if there are not enough slots.
- *assume-valid* is loaded if possible, but falls back to *Normal* if there are not enough slots.
- *skip-worktree* is loaded if possible, but falls back to *Normal* if there are not enough slots.
- *Added* is loaded if possible, but falls back to *Modified* if there are not enough slots.

Appendix F. Tips and tricks for SSH/PuTTY

F.1. Introduction

PuTTY comes with a great session management, where you can save attributes of connections (e.g. SSH key, username, port). This page describes how to make use of it - partly in form of a FAQ. For this to work, you need the PuTTY.exe-application [https://www.chiark.greenend.org.uk/~sgtatham/putty/download.html].

F.1.1. How to use sessions

One special "session" is the `Default Settings` session, where you can set default values for all new connections (e.g. a key, a default username, enable compression, force SSH version 2 or change the default port and so on).

You can also save settings for (single) SSH connections as sessions. Take one server where the SSH server only listens on a different port, then you can set up all settings and save it to e.g. `SERVERNAME`. Now you can access this saved settings by starting PuTTY and double clicking `SERVERNAME` in the saved sessions list *OR*, when using TortoiseGit, plink or other putty applications, the entered host name/destination (e.g. `git@`*`SERVERNAME`*`:/test.git`) will be matched against the saved sessions list and if found, the settings of the saved session are used.

Many people like to use Pageant for storing all their keys. Because a PuTTY session is capable of storing a key, you don't always need Pageant. But imagine you want to store different keys for several different servers; in that case you would have to edit the PuTTY session over and over again, depending on the server you are trying to connect with. In this situation Pageant makes perfect sense, because when PuTTY, Plink, TortoiseGitPlink or any other PuTTY-based tool is trying to connect to an SSH server, it checks all private keys that Pageant holds to initiate the connection.

F.2. FAQ and examples section

This section is based on the descriptions above and will bring some examples for the usage with TortoiseGit (and plink).

The examples assume that you want to clone `git@example.com:/test.git`.

F.2.1. How to use a default key for all SSH connections

Start PuTTY, go to Connection->SSH->Auth and select your key. Then go to Session, select Default Settings and hit Save.

Now PuTTY (TortoiseGit and plink) will try to use this key for all new connections (no need to configure it in TortoiseGit). If the PuTTY agent is running, putty and plink try to use an already loaded key, but will ask for the password themselves (as a fallback).

F.2.2. How to connect to a SSH server on a different port

F.2.2.1. All connections to a server should use the different port

Start PuTTY, fill in the remote host name (*example.com* here) in the Host Name-field and into the Saved Sessions field. Change the port number to the number you need and click on Save. Now, when TortoiseGit/plink uses this host name and the port is automatically loaded from the session.

F.2.2.2. One special connection should use a different port

Start PuTTY, fill in the remote host name (*example.com* here) in the Host Name-field and put the remote host name followed by e.g. a number into the Saved Sessions field (e.g. *example.com1* or whatever you like). Change the port number to the number you need and click on Save.

Now, when you want to use this saved session use *example.com1* as the remote host name: Clone *git@example.com1:/test.git*. Plink detects that this is a saved session and loads the stored remote host name and port from the session.

You can create several sessions for a server with different session names, but make sure you do not use the remote host name (*example.com* here) as the exact session name, otherwise these settings will be the default ones if you try to connect to the server (*example.com*).

F.2.3. How to use two different SSH keys for the same user on the same host

Start PuTTY, fill in the remote host name (*example.com* here) in the Host Name-field and put the remote host name followed by e.g. a number into the Saved Sessions field (e.g. *example.com1* or whatever you like). Go to Connection->SSH->Auth and select the key which should be used for this connection. Now go back to Session and hit Save.

Now, when you want to use this saved session use *example.com1* as the remote host name: Clone *git@example.com1:/test.git*. Plink detects that this is a saved session and loads the stored remote host name and SSH key from the session.

Appendix G. Git Official Documentation

The up to date Git reference can be found on https://git-scm.com/docs/

G.1. Git User Manual

G.1.1. Git User Manual

2024-07-02

Revision History

2024-07-02

1. Introduction

Git is a fast distributed revision control system.

This manual is designed to be readable by someone with basic UNIX command-line skills, but no previous knowledge of Git.

Section 2, “Repositories and Branches” and Section 3, “Exploring Git history” explain how to fetch and study a project using git--read these chapters to learn how to build and test a particular version of a software project, search for regressions, and so on.

People needing to do actual development will also want to read Section 4, “Developing with Git” and Section 5, “Sharing development with others”.

Further chapters cover more specialized topics.

Comprehensive reference documentation is available through the man pages, or Section G.3.63, “git-help(1)” command. For example, for the command *git clone <repo>*, you can either use:

$ man git-clone

or:

$ git help clone

With the latter, you can use the manual viewer of your choice; see Section G.3.63, “git-help(1)” for more information.

See also Section G.1.1.1, “Git Quick Reference” for a brief overview of Git commands, without any explanation.

Finally, see Section G.1.1.2, “Notes and todo list for this manual” for ways that you can help make this manual more complete.

2. Repositories and Branches

2.1. How to get a Git repository

It will be useful to have a Git repository to experiment with as you read this manual.

The best way to get one is by using the Section G.3.24, “git-clone(1)” command to download a copy of an existing repository. If you don't already have a project in mind, here are some interesting examples:

```
    # Git itself (approx. 40MB download):
$ git clone git://git.kernel.org/pub/scm/git/git.git
    # the Linux kernel (approx. 640MB download):
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

The initial clone may be time-consuming for a large project, but you will only need to clone once.

The clone command creates a new directory named after the project (*git* or *linux* in the examples above). After you cd into this directory, you will see that it contains a copy of the project files, called the working tree, together with a special top-level directory named *.git*, which contains all the information about the history of the project.

2.2. How to check out a different version of a project

Git is best thought of as a tool for storing the history of a collection of files. It stores the history as a compressed collection of interrelated snapshots of the project's contents. In Git each such version is called a commit.

Those snapshots aren't necessarily all arranged in a single line from oldest to newest; instead, work may simultaneously proceed along parallel lines of development, called branches, which may merge and diverge.

A single Git repository can track development on multiple branches. It does this by keeping a list of heads which reference the latest commit on each branch; the Section G.3.10, “git-branch(1)” command shows you the list of branch heads:

```
$ git branch
* master
```

A freshly cloned repository contains a single branch head, by default named "master", with the working directory initialized to the state of the project referred to by that branch head.

Most projects also use tags. Tags, like heads, are references into the project's history, and can be listed using the Section G.3.144, “git-tag(1)” command:

```
$ git tag -l
v2.6.11
v2.6.11-tree
v2.6.12
v2.6.12-rc2
v2.6.12-rc3
v2.6.12-rc4
v2.6.12-rc5
v2.6.12-rc6
v2.6.13
...
```

Tags are expected to always point at the same version of a project, while heads are expected to advance as development progresses.

Create a new branch head pointing to one of these versions and check it out using Section G.3.140, “git-switch(1)”:

```
$ git switch -c new v2.6.13
```

The working directory then reflects the contents that the project had when it was tagged v2.6.13, and Section G.3.10, “git-branch(1)” shows two branches, with an asterisk marking the currently checked-out branch:

```
$ git branch
  master
* new
```

If you decide that you'd rather see version 2.6.17, you can modify the current branch to point at v2.6.17 instead, with

```
$ git reset --hard v2.6.17
```

Note that if the current branch head was your only reference to a particular point in history, then resetting that branch may leave you with no way to find the history it used to point to; so use this command carefully.

2.3. Understanding History: Commits

Every change in the history of a project is represented by a commit. The Section G.3.134, “git-show(1)” command shows the most recent commit on the current branch:

```
$ git show
commit 17cf781661e6d38f737f15f53ab552f1e95960d7
Author: Linus Torvalds <torvalds@ppc970.osdl.org.(none)>
Date:   Tue Apr 19 14:11:06 2005 -0700

    Remove duplicate getenv(DB_ENVIRONMENT) call

    Noted by Tony Luck.

diff --git a/init-db.c b/init-db.c
index 65898fa..b002dc6 100644
--- a/init-db.c
+++ b/init-db.c
@@ -7,7 +7,7 @@

 int main(int argc, char **argv)
 {
-       char *sha1_dir = getenv(DB_ENVIRONMENT), *path;
+       char *sha1_dir, *path;
        int len, i;

        if (mkdir(".git", 0755) < 0) {
```

As you can see, a commit shows who made the latest change, what they did, and why.

Every commit has a 40-hexdigit id, sometimes called the "object name" or the "SHA-1 id", shown on the first line of the *git show* output. You can usually refer to a commit by a shorter name, such as a tag or a branch name, but this longer name can also be useful. Most importantly, it is a globally unique name for this commit: so if you tell somebody else the object name (for example in email), then you are guaranteed that name will refer to the same commit in their repository that it does in yours (assuming their repository has that commit at all). Since the object name is computed as a hash over the contents of the commit, you are guaranteed that the commit can never change without its name also changing.

In fact, in Section 8, “Git concepts” we shall see that everything stored in Git history, including file data and directory contents, is stored in an object with a name that is a hash of its contents.

2.3.1. Understanding history: commits, parents, and reachability

Every commit (except the very first commit in a project) also has a parent commit which shows what happened before this commit. Following the chain of parents will eventually take you back to the beginning of the project.

However, the commits do not form a simple list; Git allows lines of development to diverge and then reconverge, and the point where two lines of development reconverge is called a "merge". The commit representing a merge can therefore have more than one parent, with each parent representing the most recent commit on one of the lines of development leading to that point.

The best way to see how this works is using the Section G.4.8, “gitk(1)” command; running gitk now on a Git repository and looking for merge commits will help understand how Git organizes history.

In the following, we say that commit X is "reachable" from commit Y if commit X is an ancestor of commit Y. Equivalently, you could say that Y is a descendant of X, or that there is a chain of parents leading from commit Y to commit X.

2.3.2. Understanding history: History diagrams

We will sometimes represent Git history using diagrams like the one below. Commits are shown as "o", and the links between them with lines drawn with - / and \. Time goes left to right:

```
         o--o--o <-- Branch A
        /
```

```
o--o--o <-- master
       \
        o--o--o <-- Branch B
```

If we need to talk about a particular commit, the character "o" may be replaced with another letter or number.

2.3.3. Understanding history: What is a branch?

When we need to be precise, we will use the word "branch" to mean a line of development, and "branch head" (or just "head") to mean a reference to the most recent commit on a branch. In the example above, the branch head named "A" is a pointer to one particular commit, but we refer to the line of three commits leading up to that point as all being part of "branch A".

However, when no confusion will result, we often just use the term "branch" both for branches and for branch heads.

2.4. Manipulating branches

Creating, deleting, and modifying branches is quick and easy; here's a summary of the commands:

git branch

list all branches.

git branch <branch>

create a new branch named *<branch>*, referencing the same point in history as the current branch.

git branch <branch> <start-point>

create a new branch named *<branch>*, referencing *<start-point>*, which may be specified any way you like, including using a branch name or a tag name.

git branch -d <branch>

delete the branch *<branch>*; if the branch is not fully merged in its upstream branch or contained in the current branch, this command will fail with a warning.

git branch -D <branch>

delete the branch *<branch>* irrespective of its merged status.

git switch <branch>

make the current branch *<branch>*, updating the working directory to reflect the version referenced by *<branch>*.

git switch -c <new> <start-point>

create a new branch *<new>* referencing *<start-point>*, and check it out.

The special symbol "HEAD" can always be used to refer to the current branch. In fact, Git uses a file named *HEAD* in the *.git* directory to remember which branch is current:

```
$ cat .git/HEAD
ref: refs/heads/master
```

2.5. Examining an old version without creating a new branch

The *git switch* command normally expects a branch head, but will also accept an arbitrary commit when invoked with --detach; for example, you can check out the commit referenced by a tag:

```
$ git switch --detach v2.6.17
Note: checking out 'v2.6.17'.
```

```
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another switch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command again. Example:

  git switch -c new_branch_name

HEAD is now at 427abfa Linux v2.6.17
```

The HEAD then refers to the SHA-1 of the commit instead of to a branch, and git branch shows that you are no longer on a branch:

```
$ cat .git/HEAD
427abfa28afedffadfca9dd8b067eb6d36bac53f
$ git branch
* (detached from v2.6.17)
  master
```

In this case we say that the HEAD is "detached".

This is an easy way to check out a particular version without having to make up a name for the new branch. You can still create a new branch (or tag) for this version later if you decide to.

2.6. Examining branches from a remote repository

The "master" branch that was created at the time you cloned is a copy of the HEAD in the repository that you cloned from. That repository may also have had other branches, though, and your local repository keeps branches which track each of those remote branches, called remote-tracking branches, which you can view using the *-r* option to Section G.3.10, “git-branch(1)”:

```
$ git branch -r
  origin/HEAD
  origin/html
  origin/maint
  origin/man
  origin/master
  origin/next
  origin/seen
  origin/todo
```

In this example, "origin" is called a remote repository, or "remote" for short. The branches of this repository are called "remote branches" from our point of view. The remote-tracking branches listed above were created based on the remote branches at clone time and will be updated by *git fetch* (hence *git pull*) and *git push*. See Section 2.8, “Updating a repository with git fetch” for details.

You might want to build on one of these remote-tracking branches on a branch of your own, just as you would for a tag:

```
$ git switch -c my-todo-copy origin/todo
```

You can also check out *origin/todo* directly to examine it or write a one-off patch. See detached head.

Note that the name "origin" is just the name that Git uses by default to refer to the repository that you cloned from.

2.7. Naming branches, tags, and other references

Branches, remote-tracking branches, and tags are all references to commits. All references are named with a slash-separated path name starting with *refs*; the names we've been using so far are actually shorthand:

- The branch *test* is short for *refs/heads/test*.
- The tag *v2.6.18* is short for *refs/tags/v2.6.18*.
- *origin/master* is short for *refs/remotes/origin/master*.

The full name is occasionally useful if, for example, there ever exists a tag and a branch with the same name.

(Newly created refs are actually stored in the *.git/refs* directory, under the path given by their name. However, for efficiency reasons they may also be packed together in a single file; see Section G.3.98, “git-pack-refs(1)”).

As another useful shortcut, the "HEAD" of a repository can be referred to just using the name of that repository. So, for example, "origin" is usually a shortcut for the HEAD branch in the repository "origin".

For the complete list of paths which Git checks for references, and the order it uses to decide which to choose when there are multiple references with the same shorthand name, see the "SPECIFYING REVISIONS" section of Section G.4.14, “gitrevisions(7)”.

2.8. Updating a repository with git fetch

After you clone a repository and commit a few changes of your own, you may wish to check the original repository for updates.

The *git-fetch* command, with no arguments, will update all of the remote-tracking branches to the latest version found in the original repository. It will not touch any of your own branches--not even the "master" branch that was created for you on clone.

2.9. Fetching branches from other repositories

You can also track branches from repositories other than the one you cloned from, using Section G.3.112, “git-remote(1)”:

```
$ git remote add staging git://git.kernel.org/.../gregkh/staging.git
$ git fetch staging
...
From git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/staging
 * [new branch]      master     -> staging/master
 * [new branch]      staging-linus -> staging/staging-linus
 * [new branch]      staging-next -> staging/staging-next
```

New remote-tracking branches will be stored under the shorthand name that you gave *git remote add*, in this case *staging*:

```
$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  staging/master
  staging/staging-linus
  staging/staging-next
```

If you run *git fetch* <*remote*> later, the remote-tracking branches for the named <*remote*> will be updated.

If you examine the file *.git/config*, you will see that Git has added a new stanza:

```
$ cat .git/config
...
[remote "staging"]
        url = git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/staging.git
        fetch = +refs/heads/*:refs/remotes/staging/*
...
```

This is what causes Git to track the remote's branches; you may modify or delete these configuration options by editing *.git/config* with a text editor. (See the "CONFIGURATION FILE" section of Section G.3.29, “git-config(1)” for details.)

3. Exploring Git history

Git is best thought of as a tool for storing the history of a collection of files. It does this by storing compressed snapshots of the contents of a file hierarchy, together with "commits" which show the relationships between these snapshots.

Git provides extremely flexible and fast tools for exploring the history of a project.

We start with one specialized tool that is useful for finding the commit that introduced a bug into a project.

3.1. How to use bisect to find a regression

Suppose version 2.6.18 of your project worked, but the version at "master" crashes. Sometimes the best way to find the cause of such a regression is to perform a brute-force search through the project's history to find the particular commit that caused the problem. The Section G.3.8, “git-bisect(1)” command can help you do this:

```
$ git bisect start
$ git bisect good v2.6.18
$ git bisect bad master
Bisecting: 3537 revisions left to test after this
[65934a9a028b88e83e2b0f8b36618fe503349f8e] BLOCK: Make USB storage depend on SCSI rather than selecting it [try #6]
```

If you run *git branch* at this point, you'll see that Git has temporarily moved you in "(no branch)". HEAD is now detached from any branch and points directly to a commit (with commit id 65934) that is reachable from "master" but not from v2.6.18. Compile and test it, and see whether it crashes. Assume it does crash. Then:

```
$ git bisect bad
Bisecting: 1769 revisions left to test after this
[7eff82c8b1511017ae605f0c99ac275a7e21b867] i2c-core: Drop useless bitmaskings
```

checks out an older version. Continue like this, telling Git at each stage whether the version it gives you is good or bad, and notice that the number of revisions left to test is cut approximately in half each time.

After about 13 tests (in this case), it will output the commit id of the guilty commit. You can then examine the commit with Section G.3.134, “git-show(1)”, find out who wrote it, and mail them your bug report with the commit id. Finally, run

```
$ git bisect reset
```

to return you to the branch you were on before.

Note that the version which *git bisect* checks out for you at each point is just a suggestion, and you're free to try a different version if you think it would be a good idea. For example, occasionally you may land on a commit that broke something unrelated; run

```
$ git bisect visualize
```

which will run gitk and label the commit it chose with a marker that says "bisect". Choose a safe-looking commit nearby, note its commit id, and check it out with:

```
$ git reset --hard fb47ddb2db
```

then test, run *bisect good* or *bisect bad* as appropriate, and continue.

Instead of *git bisect visualize* and then *git reset --hard fb47ddb2db*, you might just want to tell Git that you want to skip the current commit:

```
$ git bisect skip
```

In this case, though, Git may not eventually be able to tell the first bad one between some first skipped commits and a later bad commit.

There are also ways to automate the bisecting process if you have a test script that can tell a good from a bad commit. See Section G.3.8, “git-bisect(1)” for more information about this and other *git bisect* features.

3.2. Naming commits

We have seen several ways of naming commits already:

- 40-hexdigit object name
- branch name: refers to the commit at the head of the given branch
- tag name: refers to the commit pointed to by the given tag (we've seen branches and tags are special cases of references).
- HEAD: refers to the head of the current branch

There are many more; see the "SPECIFYING REVISIONS" section of the Section G.4.14, “gitrevisions(7)” man page for the complete list of ways to name revisions. Some examples:

```
$ git show fb47ddb2 # the first few characters of the object name
                    # are usually enough to specify it uniquely
$ git show HEAD^    # the parent of the HEAD commit
$ git show HEAD^^   # the grandparent
$ git show HEAD~4   # the great-great-grandparent
```

Recall that merge commits may have more than one parent; by default, ^ and ~ follow the first parent listed in the commit, but you can also choose:

```
$ git show HEAD^1   # show the first parent of HEAD
$ git show HEAD^2   # show the second parent of HEAD
```

In addition to HEAD, there are several other special names for commits:

Merges (to be discussed later), as well as operations such as *git reset*, which change the currently checked-out commit, generally set ORIG_HEAD to the value HEAD had before the current operation.

The *git fetch* operation always stores the head of the last fetched branch in FETCH_HEAD. For example, if you run *git fetch* without specifying a local branch as the target of the operation

```
$ git fetch git://example.com/proj.git theirbranch
```

the fetched commits will still be available from FETCH_HEAD.

When we discuss merges we'll also see the special name MERGE_HEAD, which refers to the other branch that we're merging in to the current branch.

The Section G.3.121, “git-rev-parse(1)” command is a low-level command that is occasionally useful for translating some name for a commit to the object name for that commit:

```
$ git rev-parse origin
e05db0fd4f31dde7005f075a84f96b360d05984b
```

3.3. Creating tags

We can also create a tag to refer to a particular commit; after running

```
$ git tag stable-1 1b2e1d63ff
```

You can use *stable-1* to refer to the commit 1b2e1d63ff.

This creates a "lightweight" tag. If you would also like to include a comment with the tag, and possibly sign it cryptographically, then you should create a tag object instead; see the Section G.3.144, “git-tag(1)” man page for details.

3.4. Browsing revisions

The Section G.3.74, “git-log(1)” command can show lists of commits. On its own, it shows all commits reachable from the parent commit; but you can also make more specific requests:

```
$ git log v2.5..        # commits since (not reachable from) v2.5
$ git log test..master  # commits reachable from master but not test
$ git log master..test  # ...reachable from test but not master
$ git log master...test # ...reachable from either test or master,
                        #    but not both
$ git log --since="2 weeks ago" # commits from the last 2 weeks
$ git log Makefile      # commits which modify Makefile
$ git log fs/           # ... which modify any file under fs/
$ git log -S'foo()'     # commits which add or remove any file data
                        # matching the string 'foo()'
```

And of course you can combine all of these; the following finds commits since v2.5 which touch the *Makefile* or any file under *fs*:

```
$ git log v2.5.. Makefile fs/
```

You can also ask git log to show patches:

```
$ git log -p
```

See the *--pretty* option in the Section G.3.74, “git-log(1)” man page for more display options.

Note that git log starts with the most recent commit and works backwards through the parents; however, since Git history can contain multiple independent lines of development, the particular order that commits are listed in may be somewhat arbitrary.

3.5. Generating diffs

You can generate diffs between any two versions using Section G.3.44, “git-diff(1)”:

```
$ git diff master..test
```

That will produce the diff between the tips of the two branches. If you'd prefer to find the diff from their common ancestor to test, you can use three dots instead of two:

```
$ git diff master...test
```

Sometimes what you want instead is a set of patches; for this you can use Section G.3.54, “git-format-patch(1)”:

```
$ git format-patch master..test
```

will generate a file with a patch for each commit reachable from test but not from master.

3.6. Viewing old file versions

You can always view an old version of a file by just checking out the correct revision first. But sometimes it is more convenient to be able to view an old version of a single file without checking anything out; this command does that:

```
$ git show v2.5:fs/locks.c
```

Before the colon may be anything that names a commit, and after it may be any path to a file tracked by Git.

3.7. Examples

3.7.1. Counting the number of commits on a branch

Suppose you want to know how many commits you've made on *mybranch* since it diverged from *origin*:

```
$ git log --pretty=oneline origin..mybranch | wc -l
```

Alternatively, you may often see this sort of thing done with the lower-level command Section G.3.120, “git-rev-list(1)”, which just lists the SHA-1's of all the given commits:

```
$ git rev-list origin..mybranch | wc -l
```

3.7.2. Check whether two branches point at the same history

Suppose you want to check whether two branches point at the same point in history.

```
$ git diff origin..master
```

will tell you whether the contents of the project are the same at the two branches; in theory, however, it's possible that the same project contents could have been arrived at by two different historical routes. You could compare the object names:

```
$ git rev-list origin
e05db0fd4f31dde7005f075a84f96b360d05984b
$ git rev-list master
e05db0fd4f31dde7005f075a84f96b360d05984b
```

Or you could recall that the ... operator selects all commits reachable from either one reference or the other but not both; so

```
$ git log origin...master
```

will return no commits when the two branches are equal.

3.7.3. Find first tagged version including a given fix

Suppose you know that the commit e05db0fd fixed a certain problem. You'd like to find the earliest tagged release that contains that fix.

Of course, there may be more than one answer--if the history branched after commit e05db0fd, then there could be multiple "earliest" tagged releases.

You could just visually inspect the commits since e05db0fd:

```
$ gitk e05db0fd..
```

or you can use Section G.3.93, “git-name-rev(1)”, which will give the commit a name based on any tag it finds pointing to one of the commit's descendants:

```
$ git name-rev --tags e05db0fd
e05db0fd tags/v1.5.0-rc1^0~23
```

The Section G.3.39, “git-describe(1)” command does the opposite, naming the revision using a tag on which the given commit is based:

```
$ git describe e05db0fd
v1.5.0-rc0-260-ge05db0f
```

but that may sometimes help you guess which tags might come after the given commit.

If you just want to verify whether a given tagged version contains a given commit, you could use Section G.3.81, “git-merge-base(1)”:

```
$ git merge-base e05db0fd v1.5.0-rc1
e05db0fd4f31dde7005f075a84f96b360d05984b
```

The merge-base command finds a common ancestor of the given commits, and always returns one or the other in the case where one is a descendant of the other; so the above output shows that e05db0fd actually is an ancestor of v1.5.0-rc1.

Alternatively, note that

```
$ git log v1.5.0-rc1..e05db0fd
```

will produce empty output if and only if v1.5.0-rc1 includes e05db0fd, because it outputs only commits that are not reachable from v1.5.0-rc1.

As yet another alternative, the Section G.3.131, “git-show-branch(1)” command lists the commits reachable from its arguments with a display on the left-hand side that indicates which arguments that commit is reachable from. So, if you run something like

```
$ git show-branch e05db0fd v1.5.0-rc0 v1.5.0-rc1 v1.5.0-rc2
! [e05db0fd] Fix warnings in sha1_file.c - use C99 printf format if
available
 ! [v1.5.0-rc0] GIT v1.5.0 preview
  ! [v1.5.0-rc1] GIT v1.5.0-rc1
   ! [v1.5.0-rc2] GIT v1.5.0-rc2
...
```

then a line like

```
+ ++ [e05db0fd] Fix warnings in sha1_file.c - use C99 printf format if
available
```

shows that e05db0fd is reachable from itself, from v1.5.0-rc1, and from v1.5.0-rc2, and not from v1.5.0-rc0.

3.7.4. Showing commits unique to a given branch

Suppose you would like to see all the commits reachable from the branch head named *master* but not from any other head in your repository.

We can list all the heads in this repository with Section G.3.133, “git-show-ref(1)”:

```
$ git show-ref --heads
bf62196b5e363d73353a9dcf094c59595f3153b7 refs/heads/core-tutorial
db768d5504c1bb46f63ee9d6e1772bd047e05bf9 refs/heads/maint
a07157ac624b2524a059a3414e99f6f44bebc1e7 refs/heads/master
24dbc180ea14dc1aebe09f14c8ecf32010690627 refs/heads/tutorial-2
1e87486ae06626c2f31eaa63d26fc0fd646c8af2 refs/heads/tutorial-fixes
```

We can get just the branch-head names, and remove *master*, with the help of the standard utilities cut and grep:

```
$ git show-ref --heads | cut -d' ' -f2 | grep -v '^refs/heads/master'
refs/heads/core-tutorial
refs/heads/maint
refs/heads/tutorial-2
refs/heads/tutorial-fixes
```

And then we can ask to see all the commits reachable from master but not from these other heads:

```
$ gitk master --not $( git show-ref --heads | cut -d' ' -f2 |
                       grep -v '^refs/heads/master' )
```

Obviously, endless variations are possible; for example, to see all commits reachable from some head but not from any tag in the repository:

```
$ gitk $( git show-ref --heads ) --not  $( git show-ref --tags )
```

(See Section G.4.14, “gitrevisions(7)” for explanations of commit-selecting syntax such as *--not*.)

3.7.5. Creating a changelog and tarball for a software release

The Section G.3.7, “git-archive(1)” command can create a tar or zip archive from any version of a project; for example:

```
$ git archive -o latest.tar.gz --prefix=project/ HEAD
```

will use HEAD to produce a gzipped tar archive in which each filename is preceded by *project/*. The output file format is inferred from the output file extension if possible, see Section G.3.7, “git-archive(1)” for details.

Versions of Git older than 1.7.7 don't know about the *tar.gz* format, you'll need to use gzip explicitly:

```
$ git archive --format=tar --prefix=project/ HEAD | gzip >latest.tar.gz
```

If you're releasing a new version of a software project, you may want to simultaneously make a changelog to include in the release announcement.

Linus Torvalds, for example, makes new kernel releases by tagging them, then running:

```
$ release-script 2.6.12 2.6.13-rc6 2.6.13-rc7
```

where release-script is a shell script that looks like:

```
#!/bin/sh
stable="$1"
last="$2"
new="$3"
echo "# git tag v$new"
echo "git archive --prefix=linux-$new/ v$new | gzip -9 > ../linux-$new.tar.gz"
echo "git diff v$stable v$new | gzip -9 > ../patch-$new.gz"
echo "git log --no-merges v$new ^v$last > ../ChangeLog-$new"
echo "git shortlog --no-merges v$new ^v$last > ../ShortLog"
echo "git diff --stat --summary -M v$last v$new > ../diffstat-$new"
```

and then he just cut-and-pastes the output commands after verifying that they look OK.

3.7.6. Finding commits referencing a file with given content

Somebody hands you a copy of a file, and asks which commits modified a file such that it contained the given content either before or after the commit. You can find out with this:

```
$  git log --raw --abbrev=40 --pretty=oneline |
      grep -B 1 `git hash-object filename`
```

Figuring out why this works is left as an exercise to the (advanced) student. The Section G.3.74, “git-log(1)”, Section G.3.43, “git-diff-tree(1)”, and Section G.3.62, “git-hash-object(1)” man pages may prove helpful.

4. Developing with Git

4.1. Telling Git your name

Before creating any commits, you should introduce yourself to Git. The easiest way to do so is to use Section G.3.29, “git-config(1)”:

```
$ git config --global user.name 'Your Name Comes Here'
$ git config --global user.email 'you@yourdomain.example.com'
```

Which will add the following to a file named *.gitconfig* in your home directory:

```
[user]
	name = Your Name Comes Here
	email = you@yourdomain.example.com
```

See the "CONFIGURATION FILE" section of Section G.3.29, “git-config(1)” for details on the configuration file. The file is plain text, so you can also edit it with your favorite editor.

4.2. Creating a new repository

Creating a new repository from scratch is very easy:

```
$ mkdir project
$ cd project
$ git init
```

If you have some initial content (say, a tarball):

```
$ tar xzvf project.tar.gz
$ cd project
$ git init
$ git add . # include everything below ./ in the first commit:
$ git commit
```

4.3. How to make a commit

Creating a new commit takes three steps:

1. Making some changes to the working directory using your favorite editor.

2. Telling Git about your changes.

3. Creating the commit using the content you told Git about in step 2.

In practice, you can interleave and repeat steps 1 and 2 as many times as you want: in order to keep track of what you want committed at step 3, Git maintains a snapshot of the tree's contents in a special staging area called "the index."

At the beginning, the content of the index will be identical to that of the HEAD. The command *git diff --cached*, which shows the difference between the HEAD and the index, should therefore produce no output at that point.

Modifying the index is easy:

To update the index with the contents of a new or modified file, use

```
$ git add path/to/file
```

To remove a file from the index and from the working tree, use

```
$ git rm path/to/file
```

After each step you can verify that

```
$ git diff --cached
```

always shows the difference between the HEAD and the index file--this is what you'd commit if you created the commit now--and that

```
$ git diff
```

shows the difference between the working tree and the index file.

Note that *git add* always adds just the current contents of a file to the index; further changes to the same file will be ignored unless you run *git add* on the file again.

When you're ready, just run

```
$ git commit
```

and Git will prompt you for a commit message and then create the new commit. Check to make sure it looks like what you expected with

```
$ git show
```

As a special shortcut,

```
$ git commit -a
```

will update the index with any files that you've modified or removed and create a commit, all in one step.

A number of commands are useful for keeping track of what you're about to commit:

```
$ git diff --cached # difference between HEAD and the index; what
                    # would be committed if you ran "commit" now.
$ git diff          # difference between the index file and your
                    # working directory; changes that would not
                    # be included if you ran "commit" now.
$ git diff HEAD     # difference between HEAD and working tree; what
                    # would be committed if you ran "commit -a" now.
$ git status        # a brief per-file summary of the above.
```

You can also use Section G.3.61, “git-gui(1)” to create commits, view changes in the index and the working tree files, and individually select diff hunks for inclusion in the index (by right-clicking on the diff hunk and choosing "Stage Hunk For Commit").

4.4. Creating good commit messages

Though not required, it's a good idea to begin the commit message with a single short (no more than 50 characters) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout Git. For example, Section G.3.54, “git-format-patch(1)” turns a commit into email, and it uses the title on the Subject line and the rest of the commit in the body.

4.5. Ignoring files

A project will often generate files that you do *not* want to track with Git. This typically includes files generated by a build process or temporary backup files made by your editor. Of course, *not* tracking files with Git is just a matter of *not* calling *git add* on them. But it quickly becomes annoying to have these untracked files lying around; e.g. they make *git add* . practically useless, and they keep showing up in the output of *git status*.

You can tell Git to ignore certain files by creating a file called *.gitignore* in the top level of your working directory, with contents such as:

```
# Lines starting with '#' are considered comments.
# Ignore any file named foo.txt.
foo.txt
# Ignore (generated) html files,
*.html
# except foo.html which is maintained by hand.
!foo.html
# Ignore objects and archives.
*.[oa]
```

See Section G.4.5, “gitignore(5)” for a detailed explanation of the syntax. You can also place .gitignore files in other directories in your working tree, and they will apply to those directories and their subdirectories. The *.gitignore* files can be added to your repository like any other files (just run *git add .gitignore* and *git commit*, as usual), which is convenient when the exclude patterns (such as patterns matching build output files) would also make sense for other users who clone your repository.

If you wish the exclude patterns to affect only certain repositories (instead of every repository for a given project), you may instead put them in a file in your repository named *.git/info/exclude*, or in any file specified by the *core.excludesFile* configuration variable. Some Git commands can also take exclude patterns directly on the command line. See Section G.4.5, “gitignore(5)” for the details.

4.6. How to merge

You can rejoin two diverging branches of development using Section G.3.86, “git-merge(1)”:

$ git merge branchname

merges the development in the branch *branchname* into the current branch.

A merge is made by combining the changes made in *branchname* and the changes made up to the latest commit in your current branch since their histories forked. The work tree is overwritten by the result of the merge when this combining is done cleanly, or overwritten by a half-merged results when this combining results in conflicts. Therefore, if you have uncommitted changes touching the same files as the ones impacted by the merge, Git will refuse to proceed. Most of the time, you will want to commit your changes before you can merge, and if you don't, then Section G.3.137, “git-stash(1)” can take these changes away while you're doing the merge, and reapply them afterwards.

If the changes are independent enough, Git will automatically complete the merge and commit the result (or reuse an existing commit in case of fast-forward, see below). On the other hand, if there are conflicts--for example, if the same file is modified in two different ways in the remote branch and the local branch--then you are warned; the output may look something like this:

```
$ git merge next
 100% (4/4) done
Auto-merged file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Conflict markers are left in the problematic files, and after you resolve the conflicts manually, you can update the index with the contents and run Git commit, as you normally would when creating a new file.

If you examine the resulting commit using gitk, you will see that it has two parents, one pointing to the top of the current branch, and one to the top of the other branch.

4.7. Resolving a merge

When a merge isn't resolved automatically, Git leaves the index and the working tree in a special state that gives you all the information you need to help resolve the merge.

Files with conflicts are marked specially in the index, so until you resolve the problem and update the index, Section G.3.28, “git-commit(1)” will fail:

```
$ git commit
file.txt: needs merge
```

Also, Section G.3.138, “git-status(1)” will list those files as "unmerged", and the files with conflicts will have conflict markers added, like this:

```
<<<<<<< HEAD:file.txt
Hello world
=======
```

```
Goodbye
>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

All you need to do is edit the files to resolve the conflicts, and then

```
$ git add file.txt
$ git commit
```

Note that the commit message will already be filled in for you with some information about the merge. Normally you can just use this default message unchanged, but you may add additional commentary of your own if desired.

The above is all you need to know to resolve a simple merge. But Git also provides more information to help resolve conflicts:

4.7.1. Getting conflict-resolution help during a merge

All of the changes that Git was able to merge automatically are already added to the index file, so Section G.3.44, “git-diff(1)” shows only the conflicts. It uses an unusual syntax:

```
$ git diff
diff --cc file.txt
index 802992c,2b60207..0000000
--- a/file.txt
+++ b/file.txt
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<<< HEAD:file.txt
 +Hello world
++=======
+ Goodbye
++>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

Recall that the commit which will be committed after we resolve this conflict will have two parents instead of the usual one: one parent will be HEAD, the tip of the current branch; the other will be the tip of the other branch, which is stored temporarily in MERGE_HEAD.

During the merge, the index holds three versions of each file. Each of these three "file stages" represents a different version of the file:

```
$ git show :1:file.txt  # the file in a common ancestor of both branches
$ git show :2:file.txt  # the version from HEAD.
$ git show :3:file.txt  # the version from MERGE_HEAD.
```

When you ask Section G.3.44, “git-diff(1)” to show the conflicts, it runs a three-way diff between the conflicted merge results in the work tree with stages 2 and 3 to show only hunks whose contents come from both sides, mixed (in other words, when a hunk's merge results come only from stage 2, that part is not conflicting and is not shown. Same for stage 3).

The diff above shows the differences between the working-tree version of file.txt and the stage 2 and stage 3 versions. So instead of preceding each line by a single + or -, it now uses two columns: the first column is used for differences between the first parent and the working directory copy, and the second for differences between the second parent and the working directory copy. (See the "COMBINED DIFF FORMAT" section of Section G.3.41, “git-diff-files(1)” for a details of the format.)

After resolving the conflict in the obvious way (but before updating the index), the diff will look like:

```
$ git diff
diff --cc file.txt
index 802992c,2b60207..0000000
--- a/file.txt
+++ b/file.txt
```

```
@@@ -1,1 -1,1 +1,1 @@@
- Hello world
 -Goodbye
++Goodbye world
```

This shows that our resolved version deleted "Hello world" from the first parent, deleted "Goodbye" from the second parent, and added "Goodbye world", which was previously absent from both.

Some special diff options allow diffing the working directory against any of these stages:

```
$ git diff -1 file.txt         # diff against stage 1
$ git diff --base file.txt     # same as the above
$ git diff -2 file.txt         # diff against stage 2
$ git diff --ours file.txt     # same as the above
$ git diff -3 file.txt         # diff against stage 3
$ git diff --theirs file.txt   # same as the above.
```

When using the *ort* merge strategy (the default), before updating the working tree with the result of the merge, Git writes a ref named AUTO_MERGE reflecting the state of the tree it is about to write. Conflicted paths with textual conflicts that could not be automatically merged are written to this tree with conflict markers, just as in the working tree. AUTO_MERGE can thus be used with Section G.3.44, “git-diff(1)” to show the changes you've made so far to resolve conflicts. Using the same example as above, after resolving the conflict we get:

```
$ git diff AUTO_MERGE
diff --git a/file.txt b/file.txt
index cd10406..8bf5ae7 100644
--- a/file.txt
+++ b/file.txt
@@ -1,5 +1 @@
-<<<<<<< HEAD:file.txt
-Hello world
-=======
-Goodbye
->>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
+Goodbye world
```

Notice that the diff shows we deleted the conflict markers and both versions of the content line, and wrote "Goodbye world" instead.

The Section G.3.74, “git-log(1)” and Section G.4.8, “gitk(1)” commands also provide special help for merges:

```
$ git log --merge
$ gitk --merge
```

These will display all commits which exist only on HEAD or on MERGE_HEAD, and which touch an unmerged file.

You may also use Section G.3.88, “git-mergetool(1)”, which lets you merge the unmerged files using external tools such as Emacs or kdiff3.

Each time you resolve the conflicts in a file and update the index:

```
$ git add file.txt
```

the different stages of that file will be "collapsed", after which *git diff* will (by default) no longer show diffs for that file.

4.8. Undoing a merge

If you get stuck and decide to just give up and throw the whole mess away, you can always return to the pre-merge state with

$ git merge --abort

Or, if you've already committed the merge that you want to throw away,

$ git reset --hard ORIG_HEAD

However, this last command can be dangerous in some cases--never throw away a commit you have already committed if that commit may itself have been merged into another branch, as doing so may confuse further merges.

4.9. Fast-forward merges

There is one special case not mentioned above, which is treated differently. Normally, a merge results in a merge commit, with two parents, one pointing at each of the two lines of development that were merged.

However, if the current branch is an ancestor of the other--so every commit present in the current branch is already contained in the other branch--then Git just performs a "fast-forward"; the head of the current branch is moved forward to point at the head of the merged-in branch, without any new commits being created.

4.10. Fixing mistakes

If you've messed up the working tree, but haven't yet committed your mistake, you can return the entire working tree to the last committed state with

$ git restore --staged --worktree :/

If you make a commit that you later wish you hadn't, there are two fundamentally different ways to fix the problem:

1. You can create a new commit that undoes whatever was done by the old commit. This is the correct thing if your mistake has already been made public.
2. You can go back and modify the old commit. You should never do this if you have already made the history public; Git does not normally expect the "history" of a project to change, and cannot correctly perform repeated merges from a branch that has had its history changed.

4.10.1. Fixing a mistake with a new commit

Creating a new commit that reverts an earlier change is very easy; just pass the Section G.3.122, “git-revert(1)” command a reference to the bad commit; for example, to revert the most recent commit:

$ git revert HEAD

This will create a new commit which undoes the change in HEAD. You will be given a chance to edit the commit message for the new commit.

You can also revert an earlier change, for example, the next-to-last:

$ git revert HEAD^

In this case Git will attempt to undo the old change while leaving intact any changes made since then. If more recent changes overlap with the changes to be reverted, then you will be asked to fix conflicts manually, just as in the case of resolving a merge.

4.10.2. Fixing a mistake by rewriting history

If the problematic commit is the most recent commit, and you have not yet made that commit public, then you may just destroy it using *git reset*.

Alternatively, you can edit the working directory and update the index to fix your mistake, just as if you were going to create a new commit, then run

$ git commit --amend

which will replace the old commit by a new commit incorporating your changes, giving you a chance to edit the old commit message first.

Again, you should never do this to a commit that may already have been merged into another branch; use Section G.3.122, “git-revert(1)” instead in that case.

It is also possible to replace commits further back in the history, but this is an advanced topic to be left for another chapter.

4.10.3. Checking out an old version of a file

In the process of undoing a previous bad change, you may find it useful to check out an older version of a particular file using Section G.3.119, “git-restore(1)”. The command

```
$ git restore --source=HEAD^ path/to/file
```

replaces path/to/file by the contents it had in the commit HEAD^, and also updates the index to match. It does not change branches.

If you just want to look at an old version of the file, without modifying the working directory, you can do that with Section G.3.134, “git-show(1)”:

```
$ git show HEAD^:path/to/file
```

which will display the given version of the file.

4.10.4. Temporarily setting aside work in progress

While you are in the middle of working on something complicated, you find an unrelated but obvious and trivial bug. You would like to fix it before continuing. You can use Section G.3.137, “git-stash(1)” to save the current state of your work, and after fixing the bug (or, optionally after doing so on a different branch and then coming back), unstash the work-in-progress changes.

```
$ git stash push -m "work in progress for foo feature"
```

This command will save your changes away to the *stash*, and reset your working tree and the index to match the tip of your current branch. Then you can make your fix as usual.

```
... edit and test ...
$ git commit -a -m "blorpl: typofix"
```

After that, you can go back to what you were working on with *git stash pop*:

```
$ git stash pop
```

4.11. Ensuring good performance

On large repositories, Git depends on compression to keep the history information from taking up too much space on disk or in memory. Some Git commands may automatically run Section G.3.58, “git-gc(1)”, so you don't have to worry about running it manually. However, compressing a large repository may take a while, so you may want to call *gc* explicitly to avoid automatic compression kicking in when it is not convenient.

4.12. Ensuring reliability

4.12.1. Checking the repository for corruption

The Section G.3.56, “git-fsck(1)” command runs a number of self-consistency checks on the repository, and reports on any problems. This may take some time.

```
$ git fsck
dangling commit 7281251ddd2a61e38657c827739c57015671a6b3
```

```
dangling commit 2706a059f258c6b245f298dc4ff2ccd30ec21a63
dangling commit 13472b7c4b80851a1bc551779171dcb03655e9b5
dangling blob 218761f9d90712d37a9c5e36f406f92202db07eb
dangling commit bf093535a34a4d35731aa2bd90fe6b176302f14f
dangling commit 8e4bec7f2ddaa268bef999853c25755452100f8e
dangling tree d50bb86186bf27b681d25af89d3b5b68382e4085
dangling tree b24c2473f1fd3d91352a624795be026d64c8841f
...
```

You will see informational messages on dangling objects. They are objects that still exist in the repository but are no longer referenced by any of your branches, and can (and will) be removed after a while with *gc*. You can run *git fsck --no-dangling* to suppress these messages, and still view real errors.

4.12.2. Recovering lost changes

4.12.2.1. Reflogs

Say you modify a branch with *git reset --hard*, and then realize that the branch was the only reference you had to that point in history.

Fortunately, Git also keeps a log, called a "reflog", of all the previous values of each branch. So in this case you can still find the old history using, for example,

```
$ git log master@{1}
```

This lists the commits reachable from the previous version of the *master* branch head. This syntax can be used with any Git command that accepts a commit, not just with *git log*. Some other examples:

```
$ git show master@{2}          # See where the branch pointed 2,
$ git show master@{3}          # 3, ... changes ago.
$ gitk master@{yesterday}       # See where it pointed yesterday,
$ gitk master@{"1 week ago"}    # ... or last week
$ git log --walk-reflogs master # show reflog entries for master
```

A separate reflog is kept for the HEAD, so

```
$ git show HEAD@{"1 week ago"}
```

will show what HEAD pointed to one week ago, not what the current branch pointed to one week ago. This allows you to see the history of what you've checked out.

The reflogs are kept by default for 30 days, after which they may be pruned. See Section G.3.109, “git-reflog(1)” and Section G.3.58, “git-gc(1)” to learn how to control this pruning, and see the "SPECIFYING REVISIONS" section of Section G.4.14, “gitrevisions(7)” for details.

Note that the reflog history is very different from normal Git history. While normal history is shared by every repository that works on the same project, the reflog history is not shared: it tells you only about how the branches in your local repository have changed over time.

4.12.2.2. Examining dangling objects

In some situations the reflog may not be able to save you. For example, suppose you delete a branch, then realize you need the history it contained. The reflog is also deleted; however, if you have not yet pruned the repository, then you may still be able to find the lost commits in the dangling objects that *git fsck* reports. See Section 8.1.7, “Dangling objects” for the details.

```
$ git fsck
dangling commit 7281251ddd2a61e38657c827739c57015671a6b3
dangling commit 2706a059f258c6b245f298dc4ff2ccd30ec21a63
dangling commit 13472b7c4b80851a1bc551779171dcb03655e9b5
```

...

You can examine one of those dangling commits with, for example,

$ gitk 7281251ddd --not --all

which does what it sounds like: it says that you want to see the commit history that is described by the dangling commit(s), but not the history that is described by all your existing branches and tags. Thus you get exactly the history reachable from that commit that is lost. (And notice that it might not be just one commit: we only report the "tip of the line" as being dangling, but there might be a whole deep and complex commit history that was dropped.)

If you decide you want the history back, you can always create a new reference pointing to it, for example, a new branch:

$ git branch recovered-branch 7281251ddd

Other types of dangling objects (blobs and trees) are also possible, and dangling objects can arise in other situations.

5. Sharing development with others

5.1. Getting updates with git pull

After you clone a repository and commit a few changes of your own, you may wish to check the original repository for updates and merge them into your own work.

We have already seen how to keep remote-tracking branches up to date with Section G.3.49, “git-fetch(1)”, and how to merge two branches. So you can merge in changes from the original repository's master branch with:

$ git fetch
$ git merge origin/master

However, the Section G.3.102, “git-pull(1)” command provides a way to do this in one step:

$ git pull origin master

In fact, if you have *master* checked out, then this branch has been configured by *git clone* to get changes from the HEAD branch of the origin repository. So often you can accomplish the above with just a simple

$ git pull

This command will fetch changes from the remote branches to your remote-tracking branches *origin/**, and merge the default branch into the current branch.

More generally, a branch that is created from a remote-tracking branch will pull by default from that branch. See the descriptions of the *branch.<name>.remote* and *branch.<name>.merge* options in Section G.3.29, “git-config(1)”, and the discussion of the *--track* option in Section G.3.19, “git-checkout(1)”, to learn how to control these defaults.

In addition to saving you keystrokes, *git pull* also helps you by producing a default commit message documenting the branch and repository that you pulled from.

(But note that no such commit will be created in the case of a fast-forward; instead, your branch will just be updated to point to the latest commit from the upstream branch.)

The *git pull* command can also be given . as the "remote" repository, in which case it just merges in a branch from the current repository; so the commands

$ git pull . branch
$ git merge branch

are roughly equivalent.

5.2. Submitting patches to a project

If you just have a few changes, the simplest way to submit them may just be to send them as patches in email:

First, use Section G.3.54, “git-format-patch(1)”; for example:

$ git format-patch origin

will produce a numbered series of files in the current directory, one for each patch in the current branch but not in *origin/HEAD*.

git format-patch can include an initial "cover letter". You can insert commentary on individual patches after the three dash line which *format-patch* places after the commit message but before the patch itself. If you use *git notes* to track your cover letter material, *git format-patch --notes* will include the commit's notes in a similar manner.

You can then import these into your mail client and send them by hand. However, if you have a lot to send at once, you may prefer to use the Section G.3.124, “git-send-email(1)” script to automate the process. Consult the mailing list for your project first to determine their requirements for submitting patches.

5.3. Importing patches to a project

Git also provides a tool called Section G.3.3, “git-am(1)” (am stands for "apply mailbox"), for importing such an emailed series of patches. Just save all of the patch-containing messages, in order, into a single mailbox file, say *patches.mbox*, then run

$ git am -3 patches.mbox

Git will apply each patch in order; if any conflicts are found, it will stop, and you can fix the conflicts as described in "Resolving a merge". (The *-3* option tells Git to perform a merge; if you would prefer it just to abort and leave your tree and index untouched, you may omit that option.)

Once the index is updated with the results of the conflict resolution, instead of creating a new commit, just run

$ git am --continue

and Git will create the commit for you and continue applying the remaining patches from the mailbox.

The final result will be a series of commits, one for each patch in the original mailbox, with authorship and commit log message each taken from the message containing each patch.

5.4. Public Git repositories

Another way to submit changes to a project is to tell the maintainer of that project to pull the changes from your repository using Section G.3.102, “git-pull(1)”. In the section "Getting updates with *git pull*" we described this as a way to get updates from the "main" repository, but it works just as well in the other direction.

If you and the maintainer both have accounts on the same machine, then you can just pull changes from each other's repositories directly; commands that accept repository URLs as arguments will also accept a local directory name:

$ git clone /path/to/repository
$ git pull /path/to/other/repository

or an ssh URL:

$ git clone ssh://yourhost/~you/repository

For projects with few developers, or for synchronizing a few private repositories, this may be all you need.

However, the more common way to do this is to maintain a separate public repository (usually on a different host) for others to pull changes from. This is usually more convenient, and allows you to cleanly separate private work in progress from publicly visible work.

You will continue to do your day-to-day work in your personal repository, but periodically "push" changes from your personal repository into your public repository, allowing other developers to pull from that repository. So the flow of changes, in a situation where there is one other developer with a public repository, looks like this:

```
                      you push
your personal repo ------------------> your public repo
      ^                                     |
      |                                     |
      | you pull                            | they pull
      |                                     |
      |                                     |
      |               they push             V
their public repo <------------------- their repo
```

We explain how to do this in the following sections.

5.4.1. Setting up a public repository

Assume your personal repository is in the directory *~/proj*. We first create a new clone of the repository and tell *git daemon* that it is meant to be public:

$ git clone --bare ~/proj proj.git
$ touch proj.git/git-daemon-export-ok

The resulting directory proj.git contains a "bare" git repository--it is just the contents of the *.git* directory, without any files checked out around it.

Next, copy *proj.git* to the server where you plan to host the public repository. You can use scp, rsync, or whatever is most convenient.

5.4.2. Exporting a Git repository via the Git protocol

This is the preferred method.

If someone else administers the server, they should tell you what directory to put the repository in, and what *git://* URL it will appear at. You can then skip to the section "Pushing changes to a public repository", below.

Otherwise, all you need to do is start Section G.3.38, “git-daemon(1)”; it will listen on port 9418. By default, it will allow access to any directory that looks like a Git directory and contains the magic file git-daemon-export-ok. Passing some directory paths as *git daemon* arguments will further restrict the exports to those paths.

You can also run *git daemon* as an inetd service; see the Section G.3.38, “git-daemon(1)” man page for details. (See especially the examples section.)

5.4.3. Exporting a git repository via HTTP

The Git protocol gives better performance and reliability, but on a host with a web server set up, HTTP exports may be simpler to set up.

All you need to do is place the newly created bare Git repository in a directory that is exported by the web server, and make some adjustments to give web clients some extra information they need:

$ mv proj.git /home/you/public_html/proj.git
$ cd proj.git
$ git --bare update-server-info
$ mv hooks/post-update.sample hooks/post-update

(For an explanation of the last two lines, see Section G.3.149, “git-update-server-info(1)” and Section G.4.7, “githooks(5)”.)

Advertise the URL of *proj.git*. Anybody else should then be able to clone or pull from that URL, for example with a command line like:

```
$ git clone http://yourserver.com/~you/proj.git
```

(See also *setup-git-server-over-http* [https://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.html] for a slightly more sophisticated setup using WebDAV which also allows pushing over HTTP.)

5.4.4. Pushing changes to a public repository

Note that the two techniques outlined above (exporting via http or git) allow other maintainers to fetch your latest changes, but they do not allow write access, which you will need to update the public repository with the latest changes created in your private repository.

The simplest way to do this is using Section G.3.103, "git-push(1)" and ssh; to update the remote branch named *master* with the latest state of your branch named *master*, run

```
$ git push ssh://yourserver.com/~you/proj.git master:master
```

or just

```
$ git push ssh://yourserver.com/~you/proj.git master
```

As with *git fetch*, *git push* will complain if this does not result in a fast-forward; see the following section for details on handling this case.

Note that the target of a *push* is normally a bare repository. You can also push to a repository that has a checked-out working tree, but a push to update the currently checked-out branch is denied by default to prevent confusion. See the description of the receive.denyCurrentBranch option in Section G.3.29, "git-config(1)" for details.

As with *git fetch*, you may also set up configuration options to save typing; so, for example:

```
$ git remote add public-repo ssh://yourserver.com/~you/proj.git
```

adds the following to *.git/config*:

```
[remote "public-repo"]
      url = yourserver.com:proj.git
      fetch = +refs/heads/*:refs/remotes/example/*
```

which lets you do the same push with just

```
$ git push public-repo master
```

See the explanations of the *remote.<name>.url*, *branch.<name>.remote*, and *remote.<name>.push* options in Section G.3.29, "git-config(1)" for details.

5.4.5. What to do when a push fails

If a push would not result in a fast-forward of the remote branch, then it will fail with an error like:

```
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to '...'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

This can happen, for example, if you:

- use *git reset --hard* to remove already-published commits, or
- use *git commit --amend* to replace already-published commits (as in Section 4.10.2, "Fixing a mistake by rewriting history"), or

- use *git rebase* to rebase any already-published commits (as in Section 6.2, “Keeping a patch series up to date using git rebase”).

You may force *git push* to perform the update anyway by preceding the branch name with a plus sign:

$ git push ssh://yourserver.com/~you/proj.git +master

Note the addition of the + sign. Alternatively, you can use the *-f* flag to force the remote update, as in:

$ git push -f ssh://yourserver.com/~you/proj.git master

Normally whenever a branch head in a public repository is modified, it is modified to point to a descendant of the commit that it pointed to before. By forcing a push in this situation, you break that convention. (See Section 6.7, “Problems with rewriting history”.)

Nevertheless, this is a common practice for people that need a simple way to publish a work-in-progress patch series, and it is an acceptable compromise as long as you warn other developers that this is how you intend to manage the branch.

It's also possible for a push to fail in this way when other people have the right to push to the same repository. In that case, the correct solution is to retry the push after first updating your work: either by a pull, or by a fetch followed by a rebase; see the next section and Section G.2.4, “gitcvs-migration(7)” for more.

5.4.6. Setting up a shared repository

Another way to collaborate is by using a model similar to that commonly used in CVS, where several developers with special rights all push to and pull from a single shared repository. See Section G.2.4, “gitcvs-migration(7)” for instructions on how to set this up.

However, while there is nothing wrong with Git's support for shared repositories, this mode of operation is not generally recommended, simply because the mode of collaboration that Git supports--by exchanging patches and pulling from public repositories--has so many advantages over the central shared repository:

- Git's ability to quickly import and merge patches allows a single maintainer to process incoming changes even at very high rates. And when that becomes too much, *git pull* provides an easy way for that maintainer to delegate this job to other maintainers while still allowing optional review of incoming changes.
- Since every developer's repository has the same complete copy of the project history, no repository is special, and it is trivial for another developer to take over maintenance of a project, either by mutual agreement, or because a maintainer becomes unresponsive or difficult to work with.
- The lack of a central group of "committers" means there is less need for formal decisions about who is "in" and who is "out".

5.4.7. Allowing web browsing of a repository

The gitweb cgi script provides users an easy way to browse your project's revisions, file contents and logs without having to install Git. Features like RSS/Atom feeds and blame/annotation details may optionally be enabled.

The Section G.3.72, “git-instaweb(1)” command provides a simple way to start browsing the repository using gitweb. The default server when using instaweb is lighttpd.

See the file gitweb/INSTALL in the Git source tree and Section G.4.16, “gitweb(1)” for instructions on details setting up a permanent installation with a CGI or Perl capable server.

5.5. How to get a Git repository with minimal history

A shallow clone, with its truncated history, is useful when one is interested only in recent history of a project and getting full history from the upstream is expensive.

A shallow clone is created by specifying the Section G.3.24, “git-clone(1)” *--depth* switch. The depth can later be changed with the Section G.3.49, “git-fetch(1)” *--depth* switch, or full history restored with *--unshallow*.

Merging inside a shallow clone will work as long as a merge base is in the recent history. Otherwise, it will be like merging unrelated histories and may have to result in huge conflicts. This limitation may make such a repository unsuitable to be used in merge based workflows.

5.6. Examples

5.6.1. Maintaining topic branches for a Linux subsystem maintainer

This describes how Tony Luck uses Git in his role as maintainer of the IA64 architecture for the Linux kernel.

He uses two public branches:

- A "test" tree into which patches are initially placed so that they can get some exposure when integrated with other ongoing development. This tree is available to Andrew for pulling into -mm whenever he wants.
- A "release" tree into which tested patches are moved for final sanity checking, and as a vehicle to send them upstream to Linus (by sending him a "please pull" request.)

He also uses a set of temporary branches ("topic branches"), each containing a logical grouping of patches.

To set this up, first create your work tree by cloning Linus's public tree:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git work
$ cd work
```

Linus's tree will be stored in the remote-tracking branch named origin/master, and can be updated using Section G.3.49, “git-fetch(1)”; you can track other public trees using Section G.3.112, “git-remote(1)” to set up a "remote" and Section G.3.49, “git-fetch(1)” to keep them up to date; see Section 2, “Repositories and Branches”.

Now create the branches in which you are going to work; these start out at the current tip of origin/master branch, and should be set up (using the *--track* option to Section G.3.10, “git-branch(1)”) to merge changes in from Linus by default.

```
$ git branch --track test origin/master
$ git branch --track release origin/master
```

These can be easily kept up to date using Section G.3.102, “git-pull(1)”.

```
$ git switch test && git pull
$ git switch release && git pull
```

Important note! If you have any local changes in these branches, then this merge will create a commit object in the history (with no local changes Git will simply do a "fast-forward" merge). Many people dislike the "noise" that this creates in the Linux history, so you should avoid doing this capriciously in the *release* branch, as these noisy commits will become part of the permanent history when you ask Linus to pull from the release branch.

A few configuration variables (see Section G.3.29, “git-config(1)”) can make it easy to push both branches to your public tree. (See Section 5.4.1, “Setting up a public repository”.)

```
$ cat >> .git/config <<EOF
[remote "mytree"]
	url =  master.kernel.org:/pub/scm/linux/kernel/git/aegl/linux.git
	push = release
	push = test
EOF
```

Then you can push both the test and release trees using Section G.3.103, “git-push(1)”:

```
$ git push mytree
```

or push just one of the test and release branches using:

$ git push mytree test

or

$ git push mytree release

Now to apply some patches from the community. Think of a short snappy name for a branch to hold this patch (or related group of patches), and create a new branch from a recent stable tag of Linus's branch. Picking a stable base for your branch will: 1) help you: by avoiding inclusion of unrelated and perhaps lightly tested changes 2) help future bug hunters that use *git bisect* to find problems

$ git switch -c speed-up-spinlocks v2.6.35

Now you apply the patch(es), run some tests, and commit the change(s). If the patch is a multi-part series, then you should apply each as a separate commit to this branch.

$ ... patch ... test ... commit [... patch ... test ... commit]*

When you are happy with the state of this change, you can merge it into the "test" branch in preparation to make it public:

$ git switch test && git merge speed-up-spinlocks

It is unlikely that you would have any conflicts here … but you might if you spent a while on this step and had also pulled new versions from upstream.

Sometime later when enough time has passed and testing done, you can pull the same branch into the *release* tree ready to go upstream. This is where you see the value of keeping each patch (or patch series) in its own branch. It means that the patches can be moved into the *release* tree in any order.

$ git switch release && git merge speed-up-spinlocks

After a while, you will have a number of branches, and despite the well chosen names you picked for each of them, you may forget what they are for, or what status they are in. To get a reminder of what changes are in a specific branch, use:

$ git log linux..branchname | git shortlog

To see whether it has already been merged into the test or release branches, use:

$ git log test..branchname

or

$ git log release..branchname

(If this branch has not yet been merged, you will see some log entries. If it has been merged, then there will be no output.)

Once a patch completes the great cycle (moving from test to release, then pulled by Linus, and finally coming back into your local *origin/master* branch), the branch for this change is no longer needed. You detect this when the output from:

$ git log origin..branchname

is empty. At this point the branch can be deleted:

$ git branch -d branchname

Some changes are so trivial that it is not necessary to create a separate branch and then merge into each of the test and release branches. For these changes, just apply directly to the *release* branch, and then merge that into the *test* branch.

After pushing your work to *mytree*, you can use Section G.3.116, “git-request-pull(1)” to prepare a "please pull" request message to send to Linus:

```
$ git push mytree
$ git request-pull origin mytree release
```

Here are some of the scripts that simplify all this even further.

```
==== update script ====
# Update a branch in my Git tree.  If the branch to be updated
# is origin, then pull from kernel.org.  Otherwise merge
# origin/master branch into test|release branch

case "$1" in
test|release)
	git checkout $1 && git pull . origin
	;;
origin)
	before=$(git rev-parse refs/remotes/origin/master)
	git fetch origin
	after=$(git rev-parse refs/remotes/origin/master)
	if [ $before != $after ]
	then
		git log $before..$after | git shortlog
	fi
	;;
*)
	echo "usage: $0 origin|test|release" 1>&2
	exit 1
	;;
esac

==== merge script ====
# Merge a branch into either the test or release branch

pname=$0

usage()
{
	echo "usage: $pname branch test|release" 1>&2
	exit 1
}

git show-ref -q --verify -- refs/heads/"$1" || {
	echo "Can't see branch <$1>" 1>&2
	usage
}

case "$2" in
test|release)
	if [ $(git log $2..$1 | wc -c) -eq 0 ]
	then
		echo $1 already merged into $2 1>&2
		exit 1
	fi
	git checkout $2 && git pull . $1
	;;
*)
	usage
```

```
        ;;
esac

==== status script ====
# report on status of my ia64 Git tree

gb=$(tput setab 2)
rb=$(tput setab 1)
restore=$(tput setab 9)

if [ `git rev-list test..release | wc -c` -gt 0 ]
then
        echo $rb Warning: commits in release that are not in test $restore
        git log test..release
fi

for branch in `git show-ref --heads | sed 's|^.*/||'`
do
        if [ $branch = test -o $branch = release ]
        then
                continue
        fi

        echo -n $gb ======= $branch ====== $restore " "
        status=
        for ref in test release origin/master
        do
                if [ `git rev-list $ref..$branch | wc -c` -gt 0 ]
                then
                        status=$status${ref:0:1}
                fi
        done
        case $status in
        trl)
                echo $rb Need to pull into test $restore
                ;;
        rl)
                echo "In test"
                ;;
        l)
                echo "Waiting for linus"
                ;;
        "")
                echo $rb All done $restore
                ;;
        *)
                echo $rb "<$status>" $restore
                ;;
        esac
        git log origin/master..$branch | git shortlog
done
```

6. Rewriting history and maintaining patch series

Normally commits are only added to a project, never taken away or replaced. Git is designed with this assumption, and violating it will cause Git's merge machinery (for example) to do the wrong thing.

However, there is a situation in which it can be useful to violate this assumption.

6.1. Creating the perfect patch series

Suppose you are a contributor to a large project, and you want to add a complicated feature, and to present it to the other developers in a way that makes it easy for them to read your changes, verify that they are correct, and understand why you made each change.

If you present all of your changes as a single patch (or commit), they may find that it is too much to digest all at once.

If you present them with the entire history of your work, complete with mistakes, corrections, and dead ends, they may be overwhelmed.

So the ideal is usually to produce a series of patches such that:

1. Each patch can be applied in order.
2. Each patch includes a single logical change, together with a message explaining the change.
3. No patch introduces a regression: after applying any initial part of the series, the resulting project still compiles and works, and has no bugs that it didn't have before.
4. The complete series produces the same end result as your own (probably much messier!) development process did.

We will introduce some tools that can help you do this, explain how to use them, and then explain some of the problems that can arise because you are rewriting history.

6.2. Keeping a patch series up to date using git rebase

Suppose that you create a branch *mywork* on a remote-tracking branch *origin*, and create some commits on top of it:

$ git switch -c mywork origin
$ vi file.txt
$ git commit
$ vi otherfile.txt
$ git commit
...

You have performed no merges into mywork, so it is just a simple linear sequence of patches on top of *origin*:

```
o--o--O <-- origin
       \
        a--b--c <-- mywork
```

Some more interesting work has been done in the upstream project, and *origin* has advanced:

```
o--o--O--o--o--o <-- origin
       \
        a--b--c <-- mywork
```

At this point, you could use *pull* to merge your changes back in; the result would create a new merge commit, like this:

```
o--o--O--o--o--o <-- origin
       \        \
        a--b--c--m <-- mywork
```

However, if you prefer to keep the history in mywork a simple series of commits without any merges, you may instead choose to use Section G.3.107, “git-rebase(1)”:

$ git switch mywork
$ git rebase origin

This will remove each of your commits from mywork, temporarily saving them as patches (in a directory named *.git/rebase-apply*), update mywork to point at the latest version of origin, then apply each of the saved patches to the new mywork. The result will look like:

```
o--o--O--o--o--o <-- origin
                 \
                  a'--b'--c' <-- mywork
```

In the process, it may discover conflicts. In that case it will stop and allow you to fix the conflicts; after fixing conflicts, use *git add* to update the index with those contents, and then, instead of running *git commit*, just run

$ git rebase --continue

and Git will continue applying the rest of the patches.

At any point you may use the *--abort* option to abort this process and return mywork to the state it had before you started the rebase:

$ git rebase --abort

If you need to reorder or edit a number of commits in a branch, it may be easier to use *git rebase -i*, which allows you to reorder and squash commits, as well as marking them for individual editing during the rebase. See Section 6.5, “Using interactive rebases” for details, and Section 6.4, “Reordering or selecting from a patch series” for alternatives.

6.3. Rewriting a single commit

We saw in Section 4.10.2, “Fixing a mistake by rewriting history” that you can replace the most recent commit using

$ git commit --amend

which will replace the old commit by a new commit incorporating your changes, giving you a chance to edit the old commit message first. This is useful for fixing typos in your last commit, or for adjusting the patch contents of a poorly staged commit.

If you need to amend commits from deeper in your history, you can use interactive rebase's *edit* instruction.

6.4. Reordering or selecting from a patch series

Sometimes you want to edit a commit deeper in your history. One approach is to use *git format-patch* to create a series of patches and then reset the state to before the patches:

$ git format-patch origin
$ git reset --hard origin

Then modify, reorder, or eliminate patches as needed before applying them again with Section G.3.3, “git-am(1)”:

$ git am *.patch

6.5. Using interactive rebases

You can also edit a patch series with an interactive rebase. This is the same as reordering a patch series using *format-patch*, so use whichever interface you like best.

Rebase your current HEAD on the last commit you want to retain as-is. For example, if you want to reorder the last 5 commits, use:

$ git rebase -i HEAD~5

This will open your editor with a list of steps to be taken to perform your rebase.

```
pick deadbee The oneline of this commit
pick fa1afe1 The oneline of the next commit
...

# Rebase c0ffeee..deadbee onto c0ffeee
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

As explained in the comments, you can reorder commits, squash them together, edit commit messages, etc. by editing the list. Once you are satisfied, save the list and close your editor, and the rebase will begin.

The rebase will stop where *pick* has been replaced with *edit* or when a step in the list fails to mechanically resolve conflicts and needs your help. When you are done editing and/or resolving conflicts you can continue with *git rebase --continue*. If you decide that things are getting too hairy, you can always bail out with *git rebase --abort*. Even after the rebase is complete, you can still recover the original branch by using the reflog.

For a more detailed discussion of the procedure and additional tips, see the "INTERACTIVE MODE" section of Section G.3.107, “git-rebase(1)”.

6.6. Other tools

There are numerous other tools, such as StGit, which exist for the purpose of maintaining a patch series. These are outside of the scope of this manual.

6.7. Problems with rewriting history

The primary problem with rewriting the history of a branch has to do with merging. Suppose somebody fetches your branch and merges it into their branch, with a result something like this:

```
o--o--O--o--o--o <-- origin
       \        \
        t--t--t--m <-- their branch:
```

Then suppose you modify the last three commits:

```
        o--o--o <-- new head of origin
       /
o--o--O--o--o--o <-- old head of origin
```

If we examined all this history together in one repository, it will look like:

```
        o--o--o <-- new head of origin
       /
o--o--O--o--o--o <-- old head of origin
       \        \
        t--t--t--m <-- their branch:
```

Git has no way of knowing that the new head is an updated version of the old head; it treats this situation exactly the same as it would if two developers had independently done the work on the old and new heads in parallel. At this point, if someone attempts to merge the new head in to their branch, Git will attempt to merge together the two (old and new) lines of development, instead of trying to replace the old by the new. The results are likely to be unexpected.

You may still choose to publish branches whose history is rewritten, and it may be useful for others to be able to fetch those branches in order to examine or test them, but they should not attempt to pull such branches into their own work.

For true distributed development that supports proper merging, published branches should never be rewritten.

6.8. Why bisecting merge commits can be harder than bisecting linear history

The Section G.3.8, “git-bisect(1)” command correctly handles history that includes merge commits. However, when the commit that it finds is a merge commit, the user may need to work harder than usual to figure out why that commit introduced a problem.

Imagine this history:

```
---Z---o---X---.....---o---A---C---D
    \                           /
     o---o---Y---.....---o---B
```

Suppose that on the upper line of development, the meaning of one of the functions that exists at Z is changed at commit X. The commits from Z leading to A change both the function's implementation and all calling sites that exist at Z, as well as new calling sites they add, to be consistent. There is no bug at A.

Suppose that in the meantime on the lower line of development somebody adds a new calling site for that function at commit Y. The commits from Z leading to B all assume the old semantics of that function and the callers and the callee are consistent with each other. There is no bug at B, either.

Suppose further that the two development lines merge cleanly at C, so no conflict resolution is required.

Nevertheless, the code at C is broken, because the callers added on the lower line of development have not been converted to the new semantics introduced on the upper line of development. So if all you know is that D is bad, that Z is good, and that Section G.3.8, “git-bisect(1)” identifies C as the culprit, how will you figure out that the problem is due to this change in semantics?

When the result of a *git bisect* is a non-merge commit, you should normally be able to discover the problem by examining just that commit. Developers can make this easy by breaking their changes into small self-contained commits. That won't help in the case above, however, because the problem isn't obvious from examination of any single commit; instead, a global view of the development is required. To make matters worse, the change in semantics in the problematic function may be just one small part of the changes in the upper line of development.

On the other hand, if instead of merging at C you had rebased the history between Z to B on top of A, you would have gotten this linear history:

```
---Z---o---X--.....---o---A---o---o---Y*--.....---o---B*--D*
```

Bisecting between Z and D* would hit a single culprit commit Y*, and understanding why Y* was broken would probably be easier.

Partly for this reason, many experienced Git users, even when working on an otherwise merge-heavy project, keep the history linear by rebasing against the latest upstream version before publishing.

7. Advanced branch management

7.1. Fetching individual branches

Instead of using Section G.3.112, “git-remote(1)”, you can also choose just to update one branch at a time, and to store it locally under an arbitrary name:

$ git fetch origin todo:my-todo-work

The first argument, *origin*, just tells Git to fetch from the repository you originally cloned from. The second argument tells Git to fetch the branch named *todo* from the remote repository, and to store it locally under the name *refs/heads/my-todo-work*.

You can also fetch branches from other repositories; so

$ git fetch git://example.com/proj.git master:example-master

will create a new branch named *example-master* and store in it the branch named *master* from the repository at the given URL. If you already have a branch named example-master, it will attempt to fast-forward to the commit given by example.com's master branch. In more detail:

7.2. git fetch and fast-forwards

In the previous example, when updating an existing branch, *git fetch* checks to make sure that the most recent commit on the remote branch is a descendant of the most recent commit on your copy of the branch before updating your copy of the branch to point at the new commit. Git calls this process a fast-forward.

A fast-forward looks something like this:

```
o--o--o--o <-- old head of the branch
          \
           o--o--o <-- new head of the branch
```

In some cases it is possible that the new head will **not** actually be a descendant of the old head. For example, the developer may have realized a serious mistake was made and decided to backtrack, resulting in a situation like:

```
o--o--o--o--a--b <-- old head of the branch
          \
           o--o--o <-- new head of the branch
```

In this case, *git fetch* will fail, and print out a warning.

In that case, you can still force Git to update to the new head, as described in the following section. However, note that in the situation above this may mean losing the commits labeled *a* and *b*, unless you've already created a reference of your own pointing to them.

7.3. Forcing git fetch to do non-fast-forward updates

If git fetch fails because the new head of a branch is not a descendant of the old head, you may force the update with:

$ git fetch git://example.com/proj.git +master:refs/remotes/example/master

Note the addition of the + sign. Alternatively, you can use the *-f* flag to force updates of all the fetched branches, as in:

$ git fetch -f origin

Be aware that commits that the old version of example/master pointed at may be lost, as we saw in the previous section.

7.4. Configuring remote-tracking branches

We saw above that *origin* is just a shortcut to refer to the repository that you originally cloned from. This information is stored in Git configuration variables, which you can see using Section G.3.29, “git-config(1)”:

$ git config -l
core.repositoryformatversion=0
core.filemode=true
core.logallrefupdates=true

```
remote.origin.url=git://git.kernel.org/pub/scm/git/git.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

If there are other repositories that you also use frequently, you can create similar configuration options to save typing; for example,

```
$ git remote add example git://example.com/proj.git
```

adds the following to *.git/config*:

```
[remote "example"]
	url = git://example.com/proj.git
	fetch = +refs/heads/*:refs/remotes/example/*
```

Also note that the above configuration can be performed by directly editing the file *.git/config* instead of using Section G.3.112, “git-remote(1)”.

After configuring the remote, the following three commands will do the same thing:

```
$ git fetch git://example.com/proj.git +refs/heads/*:refs/remotes/example/*
$ git fetch example +refs/heads/*:refs/remotes/example/*
$ git fetch example
```

See Section G.3.29, “git-config(1)” for more details on the configuration options mentioned above and Section G.3.49, “git-fetch(1)” for more details on the refspec syntax.

8. Git concepts

Git is built on a small number of simple but powerful ideas. While it is possible to get things done without understanding them, you will find Git much more intuitive if you do.

We start with the most important, the object database and the index.

8.1. The Object Database

We already saw in Section 2.3, “Understanding History: Commits” that all commits are stored under a 40-digit "object name". In fact, all the information needed to represent the history of a project is stored in objects with such names. In each case the name is calculated by taking the SHA-1 hash of the contents of the object. The SHA-1 hash is a cryptographic hash function. What that means to us is that it is impossible to find two different objects with the same name. This has a number of advantages; among others:

- Git can quickly determine whether two objects are identical or not, just by comparing names.
- Since object names are computed the same way in every repository, the same content stored in two repositories will always be stored under the same name.
- Git can detect errors when it reads an object, by checking that the object's name is still the SHA-1 hash of its contents.

(See Section 11.1, “Object storage format” for the details of the object formatting and SHA-1 calculation.)

There are four different types of objects: "blob", "tree", "commit", and "tag".

- A "blob" object is used to store file data.
- A "tree" object ties one or more "blob" objects into a directory structure. In addition, a tree object can refer to other tree objects, thus creating a directory hierarchy.
- A "commit" object ties such directory hierarchies together into a directed acyclic graph of revisions--each commit contains the object name of exactly one tree designating the directory hierarchy at the time of the commit.

In addition, a commit refers to "parent" commit objects that describe the history of how we arrived at that directory hierarchy.

- A "tag" object symbolically identifies and can be used to sign other objects. It contains the object name and type of another object, a symbolic name (of course!) and, optionally, a signature.

The object types in some more detail:

8.1.1. Commit Object

The "commit" object links a physical state of a tree with a description of how we got there and why. Use the *--pretty=raw* option to Section G.3.134, “git-show(1)” or Section G.3.74, “git-log(1)” to examine your favorite commit:

```
$ git show -s --pretty=raw 2be7fcb476
commit 2be7fcb4764f2dbcee52635b91fedb1b3dcf7ab4
tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf
parent 257a84d9d02e90447b149af58b271c19405edb6a
author Dave Watson <dwatson@mimvista.com> 1187576872 -0400
committer Junio C Hamano <gitster@pobox.com> 1187591163 -0700

    Fix misspelling of 'suppress' in docs

    Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

As you can see, a commit is defined by:

- a tree: The SHA-1 name of a tree object (as defined below), representing the contents of a directory at a certain point in time.
- parent(s): The SHA-1 name(s) of some number of commits which represent the immediately previous step(s) in the history of the project. The example above has one parent; merge commits may have more than one. A commit with no parents is called a "root" commit, and represents the initial revision of a project. Each project must have at least one root. A project can also have multiple roots, though that isn't common (or necessarily a good idea).
- an author: The name of the person responsible for this change, together with its date.
- a committer: The name of the person who actually created the commit, with the date it was done. This may be different from the author, for example, if the author was someone who wrote a patch and emailed it to the person who used it to create the commit.
- a comment describing this commit.

Note that a commit does not itself contain any information about what actually changed; all changes are calculated by comparing the contents of the tree referred to by this commit with the trees associated with its parents. In particular, Git does not attempt to record file renames explicitly, though it can identify cases where the existence of the same file data at changing paths suggests a rename. (See, for example, the *-M* option to Section G.3.44, “git-diff(1)”).

A commit is usually created by Section G.3.28, “git-commit(1)”, which creates a commit whose parent is normally the current HEAD, and whose tree is taken from the content currently stored in the index.

8.1.2. Tree Object

The ever-versatile Section G.3.134, “git-show(1)” command can also be used to examine tree objects, but Section G.3.77, “git-ls-tree(1)” will give you more details:

```
$ git ls-tree fb3a8bdd0ce
100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c    .gitignore
100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d    .mailmap
```

```
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    COPYING
040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745    Documentation
100755 blob 3c0032cec592a765692234f1cba47dfdcc3a9200    GIT-VERSION-GEN
100644 blob 289b046a443c0647624607d471289b2c7dcd470b    INSTALL
100644 blob 4eb463797adc693dc168b926b6932ff53f17d0b1    Makefile
100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52    README
...
```

As you can see, a tree object contains a list of entries, each with a mode, object type, SHA-1 name, and name, sorted by name. It represents the contents of a single directory tree.

The object type may be a blob, representing the contents of a file, or another tree, representing the contents of a subdirectory. Since trees and blobs, like all other objects, are named by the SHA-1 hash of their contents, two trees have the same SHA-1 name if and only if their contents (including, recursively, the contents of all subdirectories) are identical. This allows Git to quickly determine the differences between two related tree objects, since it can ignore any entries with identical object names.

(Note: in the presence of submodules, trees may also have commits as entries. See Section 9, “Submodules” for documentation.)

Note that the files all have mode 644 or 755: Git actually only pays attention to the executable bit.

8.1.3. Blob Object

You can use Section G.3.134, “git-show(1)” to examine the contents of a blob; take, for example, the blob in the entry for *COPYING* from the tree above:

```
$ git show 6ff87c4664

 Note that the only valid version of the GPL as far as this project
 is concerned is _this_ particular version of the license (ie v2, not
 v2.2 or v3.x or whatever), unless explicitly otherwise stated.
...
```

A "blob" object is nothing but a binary blob of data. It doesn't refer to anything else or have attributes of any kind.

Since the blob is entirely defined by its data, if two files in a directory tree (or in multiple different versions of the repository) have the same contents, they will share the same blob object. The object is totally independent of its location in the directory tree, and renaming a file does not change the object that file is associated with.

Note that any tree or blob object can be examined using Section G.3.134, “git-show(1)” with the <revision>:<path> syntax. This can sometimes be useful for browsing the contents of a tree that is not currently checked out.

8.1.4. Trust

If you receive the SHA-1 name of a blob from one source, and its contents from another (possibly untrusted) source, you can still trust that those contents are correct as long as the SHA-1 name agrees. This is because the SHA-1 is designed so that it is infeasible to find different contents that produce the same hash.

Similarly, you need only trust the SHA-1 name of a top-level tree object to trust the contents of the entire directory that it refers to, and if you receive the SHA-1 name of a commit from a trusted source, then you can easily verify the entire history of commits reachable through parents of that commit, and all of those contents of the trees referred to by those commits.

So to introduce some real trust in the system, the only thing you need to do is to digitally sign just *one* special note, which includes the name of a top-level commit. Your digital signature shows others that you trust that commit, and the immutability of the history of commits tells others that they can trust the whole history.

In other words, you can easily validate a whole archive by just sending out a single email that tells the people the name (SHA-1 hash) of the top commit, and digitally sign that email using something like GPG/PGP.

To assist in this, Git also provides the tag object…

8.1.5. Tag Object

A tag object contains an object, object type, tag name, the name of the person ("tagger") who created the tag, and a message, which may contain a signature, as can be seen using Section G.3.13, “git-cat-file(1)”:

```
$ git cat-file tag v1.5.0
object 437b1b20df4b356c9342dac8d38849f24ef44f27
type commit
tag v1.5.0
tagger Junio C Hamano <junkio@cox.net> 1171411200 +0000

GIT 1.5.0
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBF0lGqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCdG4ui
nLE/L9aUXdWeTFPron96DLA=
=2E+0
-----END PGP SIGNATURE-----
```

See the Section G.3.144, “git-tag(1)” command to learn how to create and verify tag objects. (Note that Section G.3.144, “git-tag(1)” can also be used to create "lightweight tags", which are not tag objects at all, but just simple references whose names begin with *refs/tags/*).

8.1.6. How Git stores objects efficiently: pack files

Newly created objects are initially created in a file named after the object's SHA-1 hash (stored in *.git/objects*).

Unfortunately this system becomes inefficient once a project has a lot of objects. Try this on an old project:

```
$ git count-objects
6930 objects, 47620 kilobytes
```

The first number is the number of objects which are kept in individual files. The second is the amount of space taken up by those "loose" objects.

You can save space and make Git faster by moving these loose objects in to a "pack file", which stores a group of objects in an efficient compressed format; the details of how pack files are formatted can be found in Section G.5.5, “gitformat-pack(5)”.

To put the loose objects into a pack, just run git repack:

```
$ git repack
Counting objects: 6020, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6020/6020), done.
Writing objects: 100% (6020/6020), done.
Total 6020 (delta 4070), reused 0 (delta 0)
```

This creates a single "pack file" in .git/objects/pack/ containing all currently unpacked objects. You can then run

```
$ git prune
```

to remove any of the "loose" objects that are now contained in the pack. This will also remove any unreferenced objects (which may be created when, for example, you use *git reset* to remove a commit). You can verify that the loose objects are gone by looking at the *.git/objects* directory or by running

```
$ git count-objects
```

0 objects, 0 kilobytes

Although the object files are gone, any commands that refer to those objects will work exactly as they did before.

The Section G.3.58, “git-gc(1)” command performs packing, pruning, and more for you, so is normally the only high-level command you need.

8.1.7. Dangling objects

The Section G.3.56, “git-fsck(1)” command will sometimes complain about dangling objects. They are not a problem.

The most common cause of dangling objects is that you've rebased a branch, or you have pulled from somebody else who rebased a branch--see Section 6, “Rewriting history and maintaining patch series”. In that case, the old head of the original branch still exists, as does everything it pointed to. The branch pointer itself just doesn't, since you replaced it with another one.

There are also other situations that cause dangling objects. For example, a "dangling blob" may arise because you did a *git add* of a file, but then, before you actually committed it and made it part of the bigger picture, you changed something else in that file and committed that **updated** thing--the old state that you added originally ends up not being pointed to by any commit or tree, so it's now a dangling blob object.

Similarly, when the "ort" merge strategy runs, and finds that there are criss-cross merges and thus more than one merge base (which is fairly unusual, but it does happen), it will generate one temporary midway tree (or possibly even more, if you had lots of criss-crossing merges and more than two merge bases) as a temporary internal merge base, and again, those are real objects, but the end result will not end up pointing to them, so they end up "dangling" in your repository.

Generally, dangling objects aren't anything to worry about. They can even be very useful: if you screw something up, the dangling objects can be how you recover your old tree (say, you did a rebase, and realized that you really didn't want to--you can look at what dangling objects you have, and decide to reset your head to some old dangling state).

For commits, you can just use:

$ gitk <dangling-commit-sha-goes-here> --not --all

This asks for all the history reachable from the given commit but not from any branch, tag, or other reference. If you decide it's something you want, you can always create a new reference to it, e.g.,

$ git branch recovered-branch <dangling-commit-sha-goes-here>

For blobs and trees, you can't do the same, but you can still examine them. You can just do

$ git show <dangling-blob/tree-sha-goes-here>

to show what the contents of the blob were (or, for a tree, basically what the *ls* for that directory was), and that may give you some idea of what the operation was that left that dangling object.

Usually, dangling blobs and trees aren't very interesting. They're almost always the result of either being a half-way mergebase (the blob will often even have the conflict markers from a merge in it, if you have had conflicting merges that you fixed up by hand), or simply because you interrupted a *git fetch* with ^C or something like that, leaving *some* of the new objects in the object database, but just dangling and useless.

Anyway, once you are sure that you're not interested in any dangling state, you can just prune all unreachable objects:

$ git prune

and they'll be gone. (You should only run *git prune* on a quiescent repository--it's kind of like doing a filesystem fsck recovery: you don't want to do that while the filesystem is mounted. *git prune* is designed not to cause any harm in such cases of concurrent accesses to a repository but you might receive confusing or scary messages.)

8.1.8. Recovering from repository corruption

By design, Git treats data trusted to it with caution. However, even in the absence of bugs in Git itself, it is still possible that hardware or operating system errors could corrupt data.

The first defense against such problems is backups. You can back up a Git directory using clone, or just using cp, tar, or any other backup mechanism.

As a last resort, you can search for the corrupted objects and attempt to replace them by hand. Back up your repository before attempting this in case you corrupt things even more in the process.

We'll assume that the problem is a single missing or corrupted blob, which is sometimes a solvable problem. (Recovering missing trees and especially commits is **much** harder).

Before starting, verify that there is corruption, and figure out where it is with Section G.3.56, “git-fsck(1)”; this may be time-consuming.

Assume the output looks like this:

```
$ git fsck --full --no-dangling
broken link from    tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8
              to    blob 4b9458b3786228369c63936db65827de3cc06200
missing blob 4b9458b3786228369c63936db65827de3cc06200
```

Now you know that blob 4b9458b3 is missing, and that the tree 2d9263c6 points to it. If you could find just one copy of that missing blob object, possibly in some other repository, you could move it into *.git/objects/4b/9458b3...* and be done. Suppose you can't. You can still examine the tree that pointed to it with Section G.3.77, “git-ls-tree(1)”, which might output something like:

```
$ git ls-tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8
100644 blob 8d14531846b95bfa3564b58ccfb7913a034323b8    .gitignore
100644 blob ebf9bf84da0aab5ed944264a5db2a65fe3a3e883    .mailmap
100644 blob ca442d313d86dc67e0a2e5d584b465bd382cbf5c    COPYING
...
100644 blob 4b9458b3786228369c63936db65827de3cc06200    myfile
...
```

So now you know that the missing blob was the data for a file named *myfile*. And chances are you can also identify the directory--let's say it's in *somedirectory*. If you're lucky the missing copy might be the same as the copy you have checked out in your working tree at *somedirectory/myfile*; you can test whether that's right with Section G.3.62, “git-hash-object(1)”:

```
$ git hash-object -w somedirectory/myfile
```

which will create and store a blob object with the contents of somedirectory/myfile, and output the SHA-1 of that object. if you're extremely lucky it might be 4b9458b3786228369c63936db65827de3cc06200, in which case you've guessed right, and the corruption is fixed!

Otherwise, you need more information. How do you tell which version of the file has been lost?

The easiest way to do this is with:

```
$ git log --raw --all --full-history -- somedirectory/myfile
```

Because you're asking for raw output, you'll now get something like

```
commit abc
Author:
Date:
...
```

```
:100644 100644 4b9458b newsha M somedirectory/myfile


commit xyz
Author:
Date:

...
:100644 100644 oldsha 4b9458b M somedirectory/myfile
```

This tells you that the immediately following version of the file was "newsha", and that the immediately preceding version was "oldsha". You also know the commit messages that went with the change from oldsha to 4b9458b and with the change from 4b9458b to newsha.

If you've been committing small enough changes, you may now have a good shot at reconstructing the contents of the in-between state 4b9458b.

If you can do that, you can now recreate the missing object with

```
$ git hash-object -w <recreated-file>
```

and your repository is good again!

(Btw, you could have ignored the *fsck*, and started with doing a

```
$ git log --raw --all
```

and just looked for the sha of the missing object (4b9458b) in that whole thing. It's up to you--Git does **have** a lot of information, it is just missing one particular blob version.

8.2. The index

The index is a binary file (generally kept in *.git/index*) containing a sorted list of path names, each with permissions and the SHA-1 of a blob object; Section G.3.75, “git-ls-files(1)” can show you the contents of the index:

```
$ git ls-files --stage
100644 63c918c667fa005ff12ad89437f2fdc80926e21c 0       .gitignore
100644 5529b198e8d14decbe4ad99db3f7fb632de0439d 0       .mailmap
100644 6ff87c4664981e4397625791c8ea3bbb5f2279a3 0       COPYING
100644 a37b2152bd26be2c2289e1f57a292534a51a93c7 0       Documentation/.gitignore
100644 fbefe9a45b00a54b58d94d06eca48b03d40a50e0 0       Documentation/Makefile
...
100644 2511aef8d89ab52be5ec6a5e46236b4b6bcd07ea 0       xdiff/xtypes.h
100644 2ade97b2574a9f77e7ae4002a4e07a6a38e46d07 0       xdiff/xutils.c
100644 d5de8292e05e7c36c4b68857c1cf9855e3d2f70a 0       xdiff/xutils.h
```

Note that in older documentation you may see the index called the "current directory cache" or just the "cache". It has three important properties:

1. The index contains all the information necessary to generate a single (uniquely determined) tree object.

   For example, running Section G.3.28, “git-commit(1)” generates this tree object from the index, stores it in the object database, and uses it as the tree object associated with the new commit.

2. The index enables fast comparisons between the tree object it defines and the working tree.

   It does this by storing some additional data for each entry (such as the last modified time). This data is not displayed above, and is not stored in the created tree object, but it can be used to determine quickly which files in the working directory differ from what was stored in the index, and thus save Git from having to read all of the data from such files to look for changes.

3. It can efficiently represent information about merge conflicts between different tree objects, allowing each pathname to be associated with sufficient information about the trees involved that you can create a three-way merge between them.

   We saw in Section 4.7.1, “Getting conflict-resolution help during a merge” that during a merge the index can store multiple versions of a single file (called "stages"). The third column in the Section G.3.75, “git-ls-files(1)” output above is the stage number, and will take on values other than 0 for files with merge conflicts.

The index is thus a sort of temporary staging area, which is filled with a tree which you are in the process of working on.

If you blow the index away entirely, you generally haven't lost any information as long as you have the name of the tree that it described.

9. Submodules

Large projects are often composed of smaller, self-contained modules. For example, an embedded Linux distribution's source tree would include every piece of software in the distribution with some local modifications; a movie player might need to build against a specific, known-working version of a decompression library; several independent programs might all share the same build scripts.

With centralized revision control systems this is often accomplished by including every module in one single repository. Developers can check out all modules or only the modules they need to work with. They can even modify files across several modules in a single commit while moving things around or updating APIs and translations.

Git does not allow partial checkouts, so duplicating this approach in Git would force developers to keep a local copy of modules they are not interested in touching. Commits in an enormous checkout would be slower than you'd expect as Git would have to scan every directory for changes. If modules have a lot of local history, clones would take forever.

On the plus side, distributed revision control systems can much better integrate with external sources. In a centralized model, a single arbitrary snapshot of the external project is exported from its own revision control and then imported into the local revision control on a vendor branch. All the history is hidden. With distributed revision control you can clone the entire external history and much more easily follow development and re-merge local changes.

Git's submodule support allows a repository to contain, as a subdirectory, a checkout of an external project. Submodules maintain their own identity; the submodule support just stores the submodule repository location and commit ID, so other developers who clone the containing project ("superproject") can easily clone all the submodules at the same revision. Partial checkouts of the superproject are possible: you can tell Git to clone none, some or all of the submodules.

The Section G.3.141, “git-submodule(1)” command is available since Git 1.5.3. Users with Git 1.5.2 can look up the submodule commits in the repository and manually check them out; earlier versions won't recognize the submodules at all.

To see how submodule support works, create four example repositories that can be used later as a submodule:

```
$ mkdir ~/git
$ cd ~/git
$ for i in a b c d
do
    mkdir $i
    cd $i
    git init
    echo "module $i" > $i.txt
    git add $i.txt
    git commit -m "Initial commit, submodule $i"
    cd ..
done
```

Now create the superproject and add all the submodules:

```
$ mkdir super
$ cd super
$ git init
$ for i in a b c d
do
        git submodule add ~/git/$i $i
done
```

> **Note**
>
> Do not use local URLs here if you plan to publish your superproject!

See what files *git submodule* created:

```
$ ls -a
.  ..  .git  .gitmodules  a  b  c  d
```

The *git submodule add* <*repo*> <*path*> command does a couple of things:

- It clones the submodule from <*repo*> to the given <*path*> under the current directory and by default checks out the master branch.
- It adds the submodule's clone path to the Section G.4.10, “gitmodules(5)” file and adds this file to the index, ready to be committed.
- It adds the submodule's current commit ID to the index, ready to be committed.

Commit the superproject:

```
$ git commit -m "Add submodules a, b, c and d."
```

Now clone the superproject:

```
$ cd ..
$ git clone super cloned
$ cd cloned
```

The submodule directories are there, but they're empty:

```
$ ls -a a
.  ..
$ git submodule status
-d266b9873ad50488163457f025db7cdd9683d88b a
-e81d457da15309b4fef4249aba9b50187999670d b
-c1536a972b9affea0f16e0680ba87332dc059146 c
-d96249ff5d57de5de093e6baff9e0aafa5276a74 d
```

> **Note**
>
> The commit object names shown above would be different for you, but they should match the HEAD commit object names of your repositories. You can check it by running *git ls-remote ../a*.

Pulling down the submodules is a two-step process. First run *git submodule init* to add the submodule repository URLs to *.git/config*:

```
$ git submodule init
```

Now use *git submodule update* to clone the repositories and check out the commits specified in the superproject:

```
$ git submodule update
$ cd a
$ ls -a
.  ..  .git  a.txt
```

One major difference between *git submodule update* and *git submodule add* is that *git submodule update* checks out a specific commit, rather than the tip of a branch. It's like checking out a tag: the head is detached, so you're not working on a branch.

```
$ git branch
* (detached from d266b98)
  master
```

If you want to make a change within a submodule and you have a detached head, then you should create or checkout a branch, make your changes, publish the change within the submodule, and then update the superproject to reference the new commit:

```
$ git switch master
```

or

```
$ git switch -c fix-up
```

then

```
$ echo "adding a line again" >> a.txt
$ git commit -a -m "Updated the submodule from within the superproject."
$ git push
$ cd ..
$ git diff
diff --git a/a b/a
index d266b98..261dfac 160000
--- a/a
+++ b/a
@@ -1 +1 @@
-Subproject commit d266b9873ad50488163457f025db7cdd9683d88b
+Subproject commit 261dfac35cb99d380eb966e102c1197139f7fa24
$ git add a
$ git commit -m "Updated submodule a."
$ git push
```

You have to run *git submodule update* after *git pull* if you want to update submodules, too.

9.1. Pitfalls with submodules

Always publish the submodule change before publishing the change to the superproject that references it. If you forget to publish the submodule change, others won't be able to clone the repository:

```
$ cd ~/git/super/a
$ echo i added another line to this file >> a.txt
$ git commit -a -m "doing it wrong this time"
$ cd ..
$ git add a
$ git commit -m "Updated submodule a again."
$ git push
$ cd ~/git/cloned
$ git pull
$ git submodule update
```

```
error: pathspec '261dfac35cb99d380eb966e102c1197139f7fa24' did not match any file(s) known to git.
Did you forget to 'git add'?
Unable to checkout '261dfac35cb99d380eb966e102c1197139f7fa24' in submodule path 'a'
```

In older Git versions it could be easily forgotten to commit new or modified files in a submodule, which silently leads to similar problems as not pushing the submodule changes. Starting with Git 1.7.0 both *git status* and *git diff* in the superproject show submodules as modified when they contain new or modified files to protect against accidentally committing such a state. *git diff* will also add a *-dirty* to the work tree side when generating patch output or used with the *--submodule* option:

```
$ git diff
diff --git a/sub b/sub
--- a/sub
+++ b/sub
@@ -1 +1 @@
-Subproject commit 3f356705649b5d566d97ff843cf193359229a453
+Subproject commit 3f356705649b5d566d97ff843cf193359229a453-dirty
$ git diff --submodule
Submodule sub 3f35670..3f35670-dirty:
```

You also should not rewind branches in a submodule beyond commits that were ever recorded in any superproject.

It's not safe to run *git submodule update* if you've made and committed changes within a submodule without checking out a branch first. They will be silently overwritten:

```
$ cat a.txt
module a
$ echo line added from private2 >> a.txt
$ git commit -a -m "line added inside private2"
$ cd ..
$ git submodule update
Submodule path 'a': checked out 'd266b9873ad50488163457f025db7cdd9683d88b'
$ cd a
$ cat a.txt
module a
```

> **Note**
>
> The changes are still visible in the submodule's reflog.

If you have uncommitted changes in your submodule working tree, *git submodule update* will not overwrite them. Instead, you get the usual warning about not being able switch from a dirty branch.

10. Low-level Git operations

Many of the higher-level commands were originally implemented as shell scripts using a smaller core of low-level Git commands. These can still be useful when doing unusual things with Git, or just as a way to understand its inner workings.

10.1. Object access and manipulation

The Section G.3.13, “git-cat-file(1)” command can show the contents of any object, though the higher-level Section G.3.134, “git-show(1)” is usually more useful.

The Section G.3.27, “git-commit-tree(1)” command allows constructing commits with arbitrary parents and trees.

A tree can be created with Section G.3.160, “git-write-tree(1)” and its data can be accessed by Section G.3.77, “git-ls-tree(1)”. Two trees can be compared with Section G.3.43, “git-diff-tree(1)”.

A tag is created with Section G.3.89, “git-mktag(1)”, and the signature can be verified by Section G.3.155, “git-verify-tag(1)”, though it is normally simpler to use Section G.3.144, “git-tag(1)” for both.

10.2. The Workflow

High-level operations such as Section G.3.28, “git-commit(1)” and Section G.3.119, “git-restore(1)” work by moving data between the working tree, the index, and the object database. Git provides low-level operations which perform each of these steps individually.

Generally, all Git operations work on the index file. Some operations work **purely** on the index file (showing the current state of the index), but most operations move data between the index file and either the database or the working directory. Thus there are four main combinations:

10.2.1. working directory → index

The Section G.3.147, “git-update-index(1)” command updates the index with information from the working directory. You generally update the index information by just specifying the filename you want to update, like so:

$ git update-index filename

but to avoid common mistakes with filename globbing etc., the command will not normally add totally new entries or remove old entries, i.e. it will normally just update existing cache entries.

To tell Git that yes, you really do realize that certain files no longer exist, or that new files should be added, you should use the *--remove* and *--add* flags respectively.

NOTE! A *--remove* flag does *not* mean that subsequent filenames will necessarily be removed: if the files still exist in your directory structure, the index will be updated with their new status, not removed. The only thing *--remove* means is that update-index will be considering a removed file to be a valid thing, and if the file really does not exist any more, it will update the index accordingly.

As a special case, you can also do *git update-index --refresh*, which will refresh the "stat" information of each index to match the current stat information. It will *not* update the object status itself, and it will only update the fields that are used to quickly test whether an object still matches its old backing store object.

The previously introduced Section G.3.2, “git-add(1)” is just a wrapper for Section G.3.147, “git-update-index(1)”.

10.2.2. index → object database

You write your current index file to a "tree" object with the program

$ git write-tree

that doesn't come with any options--it will just write out the current index into the set of tree objects that describe that state, and it will return the name of the resulting top-level tree. You can use that tree to re-generate the index at any time by going in the other direction:

10.2.3. object database → index

You read a "tree" file from the object database, and use that to populate (and overwrite--don't do this if your index contains any unsaved state that you might want to restore later!) your current index. Normal operation is just

$ git read-tree <SHA-1 of tree>

and your index file will now be equivalent to the tree that you saved earlier. However, that is only your *index* file: your working directory contents have not been modified.

10.2.4. index → working directory

You update your working directory from the index by "checking out" files. This is not a very common operation, since normally you'd just keep your files updated, and rather than write to your working directory, you'd tell the index files about the changes in your working directory (i.e. *git update-index*).

However, if you decide to jump to a new version, or check out somebody else's version, or just restore a previous tree, you'd populate your index file with read-tree, and then you need to check out the result with

$ git checkout-index filename

or, if you want to check out all of the index, use *-a*.

NOTE! *git checkout-index* normally refuses to overwrite old files, so if you have an old version of the tree already checked out, you will need to use the *-f* flag (*before* the *-a* flag or the filename) to *force* the checkout.

Finally, there are a few odds and ends which are not purely moving from one representation to the other:

10.2.5. Tying it all together

To commit a tree you have instantiated with *git write-tree*, you'd create a "commit" object that refers to that tree and the history behind it--most notably the "parent" commits that preceded it in history.

Normally a "commit" has one parent: the previous state of the tree before a certain change was made. However, sometimes it can have two or more parent commits, in which case we call it a "merge", due to the fact that such a commit brings together ("merges") two or more previous states represented by other commits.

In other words, while a "tree" represents a particular directory state of a working directory, a "commit" represents that state in time, and explains how we got there.

You create a commit object by giving it the tree that describes the state at the time of the commit, and a list of parents:

$ git commit-tree <tree> -p <parent> [(-p <parent2>)...]

and then giving the reason for the commit on stdin (either through redirection from a pipe or file, or by just typing it at the tty).

git commit-tree will return the name of the object that represents that commit, and you should save it away for later use. Normally, you'd commit a new *HEAD* state, and while Git doesn't care where you save the note about that state, in practice we tend to just write the result to the file pointed at by *.git/HEAD*, so that we can always see what the last committed state was.

Here is a picture that illustrates how various pieces fit together:

```
                 commit-tree
                  commit obj
                    +----+
                    |    |
                    |    |
                    V    V
                 +-----------+
                 | Object DB |
                 |  Backing  |
                 |   Store   |
                 +-----------+
                       ^
           write-tree  |     |
            tree obj   |     |
                       |     |  read-tree
                       |     |  tree obj
                             V
                 +-----------+
                 |   Index   |
                 |  "cache"  |
                 +-----------+
       update-index  ^
```

```
        blob obj |    |
                 |    |
checkout-index -u |    | checkout-index
        stat     |    | blob obj
                    V
              +-----------+
              | Working   |
              | Directory |
              +-----------+
```

10.3. Examining the data

You can examine the data represented in the object database and the index with various helper tools. For every object, you can use Section G.3.13, “git-cat-file(1)” to examine details about the object:

$ git cat-file -t <objectname>

shows the type of the object, and once you have the type (which is usually implicit in where you find the object), you can use

$ git cat-file blob|tree|commit|tag <objectname>

to show its contents. NOTE! Trees have binary content, and as a result there is a special helper for showing that content, called *git ls-tree*, which turns the binary content into a more easily readable form.

It's especially instructive to look at "commit" objects, since those tend to be small and fairly self-explanatory. In particular, if you follow the convention of having the top commit name in *.git/HEAD*, you can do

$ git cat-file commit HEAD

to see what the top commit was.

10.4. Merging multiple trees

Git can help you perform a three-way merge, which can in turn be used for a many-way merge by repeating the merge procedure several times. The usual situation is that you only do one three-way merge (reconciling two lines of history) and commit the result, but if you like to, you can merge several branches in one go.

To perform a three-way merge, you start with the two commits you want to merge, find their closest common parent (a third commit), and compare the trees corresponding to these three commits.

To get the "base" for the merge, look up the common parent of two commits:

$ git merge-base <commit1> <commit2>

This prints the name of a commit they are both based on. You should now look up the tree objects of those commits, which you can easily do with

$ git cat-file commit <commitname> | head -1

since the tree object information is always the first line in a commit object.

Once you know the three trees you are going to merge (the one "original" tree, aka the common tree, and the two "result" trees, aka the branches you want to merge), you do a "merge" read into the index. This will complain if it has to throw away your old index contents, so you should make sure that you've committed those--in fact you would normally always do a merge against your last commit (which should thus match what you have in your current index anyway).

To do the merge, do

$ git read-tree -m -u <origtree> <yourtree> <targettree>

which will do all trivial merge operations for you directly in the index file, and you can just write the result out with *git write-tree*.

10.5. Merging multiple trees, continued

Sadly, many merges aren't trivial. If there are files that have been added, moved or removed, or if both branches have modified the same file, you will be left with an index tree that contains "merge entries" in it. Such an index tree can *NOT* be written out to a tree object, and you will have to resolve any such merge clashes using other tools before you can write out the result.

You can examine such index state with *git ls-files --unmerged* command. An example:

```
$ git read-tree -m $orig HEAD $target
$ git ls-files --unmerged
100644 263414f423d0e4d70dae8fe53fa34614ff3e2860 1	hello.c
100644 06fa6a24256dc7e560efa5687fa84b51f0263c3a 2	hello.c
100644 cc44c73eb783565da5831b4d820c962954019b69 3	hello.c
```

Each line of the *git ls-files --unmerged* output begins with the blob mode bits, blob SHA-1, *stage number*, and the filename. The *stage number* is Git's way to say which tree it came from: stage 1 corresponds to the *$orig* tree, stage 2 to the *HEAD* tree, and stage 3 to the *$target* tree.

Earlier we said that trivial merges are done inside *git read-tree -m*. For example, if the file did not change from *$orig* to *HEAD* or *$target*, or if the file changed from *$orig* to *HEAD* and *$orig* to *$target* the same way, obviously the final outcome is what is in *HEAD*. What the above example shows is that file *hello.c* was changed from *$orig* to *HEAD* and *$orig* to *$target* in a different way. You could resolve this by running your favorite 3-way merge program, e.g. *diff3*, *merge*, or Git's own merge-file, on the blob objects from these three stages yourself, like this:

```
$ git cat-file blob 263414f >hello.c~1
$ git cat-file blob 06fa6a2 >hello.c~2
$ git cat-file blob cc44c73 >hello.c~3
$ git merge-file hello.c~2 hello.c~1 hello.c~3
```

This would leave the merge result in *hello.c~2* file, along with conflict markers if there are conflicts. After verifying the merge result makes sense, you can tell Git what the final merge result for this file is by:

```
$ mv -f hello.c~2 hello.c
$ git update-index hello.c
```

When a path is in the "unmerged" state, running *git update-index* for that path tells Git to mark the path resolved.

The above is the description of a Git merge at the lowest level, to help you understand what conceptually happens under the hood. In practice, nobody, not even Git itself, runs *git cat-file* three times for this. There is a *git merge-index* program that extracts the stages to temporary files and calls a "merge" script on it:

```
$ git merge-index git-merge-one-file hello.c
```

and that is what higher level *git merge -s resolve* is implemented with.

11. Hacking Git

This chapter covers internal details of the Git implementation which probably only Git developers need to understand.

11.1. Object storage format

All objects have a statically determined "type" which identifies the format of the object (i.e. how it is used, and how it can refer to other objects). There are currently four different object types: "blob", "tree", "commit", and "tag".

Regardless of object type, all objects share the following characteristics: they are all deflated with zlib, and have a header that not only specifies their type, but also provides size information about the data in the object. It's worth

noting that the SHA-1 hash that is used to name the object is the hash of the original data plus this header, so *sha1sum file* does not match the object name for *file* (the earliest versions of Git hashed slightly differently but the conclusion is still the same).

The following is a short example that demonstrates how these hashes can be generated manually:

Let's assume a small text file with some simple content:

```
$ echo "Hello world" >hello.txt
```

We can now manually generate the hash Git would use for this file:

- The object we want the hash for is of type "blob" and its size is 12 bytes.
- Prepend the object header to the file content and feed this to *sha1sum*:

```
$ { printf "blob 12\0"; cat hello.txt; } | sha1sum
802992c4220de19a90767f3000a79a31b98d0df7  -
```

This manually constructed hash can be verified using *git hash-object* which of course hides the addition of the header:

```
$ git hash-object hello.txt
802992c4220de19a90767f3000a79a31b98d0df7
```

As a result, the general consistency of an object can always be tested independently of the contents or the type of the object: all objects can be validated by verifying that (a) their hashes match the content of the file and (b) the object successfully inflates to a stream of bytes that forms a sequence of *<ascii-type-without-space> + <space> + <ascii-decimal-size> + <byte\0> + <binary-object-data>*.

The structured objects can further have their structure and connectivity to other objects verified. This is generally done with the *git fsck* program, which generates a full dependency graph of all objects, and verifies their internal consistency (in addition to just verifying their superficial consistency through the hash).

11.2. A birds-eye view of Git's source code

It is not always easy for new developers to find their way through Git's source code. This section gives you a little guidance to show where to start.

A good place to start is with the contents of the initial commit, with:

```
$ git switch --detach e83c5163
```

The initial revision lays the foundation for almost everything Git has today (even though details may differ in a few places), but is small enough to read in one sitting.

Note that terminology has changed since that revision. For example, the README in that revision uses the word "changeset" to describe what we now call a commit.

Also, we do not call it "cache" any more, but rather "index"; however, the file is still called *read-cache.h*.

If you grasp the ideas in that initial commit, you should check out a more recent version and skim *read-cache-ll.h*, *object.h* and *commit.h*.

In the early days, Git (in the tradition of UNIX) was a bunch of programs which were extremely simple, and which you used in scripts, piping the output of one into another. This turned out to be good for initial development, since it was easier to test new things. However, recently many of these parts have become builtins, and some of the core has been "libified", i.e. put into libgit.a for performance, portability reasons, and to avoid code duplication.

By now, you know what the index is (and find the corresponding data structures in *read-cache-ll.h*), and that there are just a couple of object types (blobs, trees, commits and tags) which inherit their common structure from *struct*

object, which is their first member (and thus, you can cast e.g. *(struct object *)commit* to achieve the *same* as *&commit->object*, i.e. get at the object name and flags).

Now is a good point to take a break to let this information sink in.

Next step: get familiar with the object naming. Read Section 3.2, “Naming commits”. There are quite a few ways to name an object (and not only revisions!). All of these are handled in *sha1_name.c*. Just have a quick look at the function *get_sha1()*. A lot of the special handling is done by functions like *get_sha1_basic()* or the likes.

This is just to get you into the groove for the most libified part of Git: the revision walker.

Basically, the initial version of *git log* was a shell script:

```
$ git-rev-list --pretty $(git-rev-parse --default HEAD "$@") | \
      LESS=-S ${PAGER:-less}
```

What does this mean?

git rev-list is the original version of the revision walker, which *always* printed a list of revisions to stdout. It is still functional, and needs to, since most new Git commands start out as scripts using *git rev-list*.

git rev-parse is not as important any more; it was only used to filter out options that were relevant for the different plumbing commands that were called by the script.

Most of what *git rev-list* did is contained in *revision.c* and *revision.h*. It wraps the options in a struct named *rev_info*, which controls how and what revisions are walked, and more.

The original job of *git rev-parse* is now taken by the function *setup_revisions()*, which parses the revisions and the common command-line options for the revision walker. This information is stored in the struct *rev_info* for later consumption. You can do your own command-line option parsing after calling *setup_revisions()*. After that, you have to call *prepare_revision_walk()* for initialization, and then you can get the commits one by one with the function *get_revision()*.

If you are interested in more details of the revision walking process, just have a look at the first implementation of *cmd_log()*; call *git show v1.3.0~155^2~4* and scroll down to that function (note that you no longer need to call *setup_pager()* directly).

Nowadays, *git log* is a builtin, which means that it is *contained* in the command *git*. The source side of a builtin is

- a function called *cmd_<bla>*, typically defined in *builtin/<bla.c>* (note that older versions of Git used to have it in *builtin-<bla>.c* instead), and declared in *builtin.h*.
- an entry in the *commands[]* array in *git.c*, and
- an entry in *BUILTIN_OBJECTS* in the *Makefile*.

Sometimes, more than one builtin is contained in one source file. For example, *cmd_whatchanged()* and *cmd_log()* both reside in *builtin/log.c*, since they share quite a bit of code. In that case, the commands which are *not* named like the *.c* file in which they live have to be listed in *BUILT_INS* in the *Makefile*.

git log looks more complicated in C than it does in the original script, but that allows for a much greater flexibility and performance.

Here again it is a good point to take a pause.

Lesson three is: study the code. Really, it is the best way to learn about the organization of Git (after you know the basic concepts).

So, think about something which you are interested in, say, "how can I access a blob just knowing the object name of it?". The first step is to find a Git command with which you can do it. In this example, it is either *git show* or *git cat-file*.

For the sake of clarity, let's stay with *git cat-file*, because it

- is plumbing, and
- was around even in the initial commit (it literally went only through some 20 revisions as *cat-file.c*, was renamed to *builtin/cat-file.c* when made a builtin, and then saw less than 10 versions).

So, look into *builtin/cat-file.c*, search for *cmd_cat_file()* and look what it does.

```
git_config(git_default_config);
if (argc != 3)
        usage("git cat-file [-t|-s|-e|-p|<type>] <sha1>");
if (get_sha1(argv[2], sha1))
        die("Not a valid object name %s", argv[2]);
```

Let's skip over the obvious details; the only really interesting part here is the call to *get_sha1()*. It tries to interpret *argv[2]* as an object name, and if it refers to an object which is present in the current repository, it writes the resulting SHA-1 into the variable *sha1*.

Two things are interesting here:

- *get_sha1()* returns 0 on *success*. This might surprise some new Git hackers, but there is a long tradition in UNIX to return different negative numbers in case of different errors--and 0 on success.
- the variable *sha1* in the function signature of *get_sha1()* is *unsigned char **, but is actually expected to be a pointer to *unsigned char[20]*. This variable will contain the 160-bit SHA-1 of the given commit. Note that whenever a SHA-1 is passed as *unsigned char **, it is the binary representation, as opposed to the ASCII representation in hex characters, which is passed as *char **.

You will see both of these things throughout the code.

Now, for the meat:

```
case 0:
        buf = read_object_with_reference(sha1, argv[1], &size, NULL);
```

This is how you read a blob (actually, not only a blob, but any type of object). To know how the function *read_object_with_reference()* actually works, find the source code for it (something like *git grep read_object_with | grep ":[a-z]"* in the Git repository), and read the source.

To find out how the result can be used, just read on in *cmd_cat_file()*:

```
write_or_die(1, buf, size);
```

Sometimes, you do not know where to look for a feature. In many such cases, it helps to search through the output of *git log*, and then *git show* the corresponding commit.

Example: If you know that there was some test case for *git bundle*, but do not remember where it was (yes, you *could git grep bundle t/*, but that does not illustrate the point!):

$ git log --no-merges t/

In the pager (*less*), just search for "bundle", go a few lines back, and see that it is in commit 18449ab0. Now just copy this object name, and paste it into the command line

$ git show 18449ab0

Voila.

Another example: Find out what to do in order to make some script a builtin:

$ git log --no-merges --diff-filter=A builtin/*.c

You see, Git is actually the best tool to find out about the source of Git itself!

12. Git Glossary

12.1. Git explained

alternate object database

Via the alternates mechanism, a repository can inherit part of its object database from another object database, which is called an "alternate".

bare repository

A bare repository is normally an appropriately named directory with a *.git* suffix that does not have a locally checked-out copy of any of the files under revision control. That is, all of the Git administrative and control files that would normally be present in the hidden *.git* sub-directory are directly present in the *repository.git* directory instead, and no other files are present and checked out. Usually publishers of public repositories make bare repositories available.

blob object

Untyped object, e.g. the contents of a file.

branch

A "branch" is a line of development. The most recent commit on a branch is referred to as the tip of that branch. The tip of the branch is referenced by a branch head, which moves forward as additional development is done on the branch. A single Git repository can track an arbitrary number of branches, but your working tree is associated with just one of them (the "current" or "checked out" branch), and HEAD points to that branch.

cache

Obsolete for: index.

chain

A list of objects, where each object in the list contains a reference to its successor (for example, the successor of a commit could be one of its parents).

changeset

BitKeeper/cvsps speak for "commit". Since Git does not store changes, but states, it really does not make sense to use the term "changesets" with Git.

checkout

The action of updating all or part of the working tree with a tree object or blob from the object database, and updating the index and HEAD if the whole working tree has been pointed at a new branch.

cherry-picking

In SCM jargon, "cherry pick" means to choose a subset of changes out of a series of changes (typically commits) and record them as a new series of changes on top of a different codebase. In Git, this is performed by the "git cherry-pick" command to extract the change introduced by an existing commit and to record it based on the tip of the current branch as a new commit.

clean

A working tree is clean, if it corresponds to the revision referenced by the current head. Also see "dirty".

commit

As a noun: A single point in the Git history; the entire history of a project is represented as a set of interrelated commits. The word "commit" is often used by Git in the same places other revision control systems use the words "revision" or "version". Also used as a short hand for commit object.

As a verb: The action of storing a new snapshot of the project's state in the Git history, by creating a new commit representing the current state of the index and advancing HEAD to point at the new commit.

commit graph concept, representations and usage

A synonym for the DAG structure formed by the commits in the object database, referenced by branch tips, using their chain of linked commits. This structure is the definitive commit graph. The graph can be represented in other ways, e.g. the "commit-graph" file.

commit-graph file

The "commit-graph" (normally hyphenated) file is a supplemental representation of the commit graph which accelerates commit graph walks. The "commit-graph" file is stored either in the .git/objects/info directory or in the info directory of an alternate object database.

commit object

An object which contains the information about a particular revision, such as parents, committer, author, date and the tree object which corresponds to the top directory of the stored revision.

commit-ish (also committish)

A commit object or an object that can be recursively dereferenced to a commit object. The following are all commit-ishes: a commit object, a tag object that points to a commit object, a tag object that points to a tag object that points to a commit object, etc.

core Git

Fundamental data structures and utilities of Git. Exposes only limited source code management tools.

DAG

Directed acyclic graph. The commit objects form a directed acyclic graph, because they have parents (directed), and the graph of commit objects is acyclic (there is no chain which begins and ends with the same object).

dangling object

An unreachable object which is not reachable even from other unreachable objects; a dangling object has no references to it from any reference or object in the repository.

dereference

Referring to a symbolic ref: the action of accessing the reference pointed at by a symbolic ref. Recursive dereferencing involves repeating the aforementioned process on the resulting ref until a non-symbolic reference is found.

Referring to a tag object: the action of accessing the object a tag points at. Tags are recursively dereferenced by repeating the operation on the result object until the result has either a specified object type (where applicable) or any non-"tag" object type. A synonym for "recursive dereference" in the context of tags is "peel".

Referring to a commit object: the action of accessing the commit's tree object. Commits cannot be dereferenced recursively.

Unless otherwise specified, "dereferencing" as it used in the context of Git commands or protocols is implicitly recursive.

detached HEAD

Normally the HEAD stores the name of a branch, and commands that operate on the history HEAD represents operate on the history leading to the tip of the branch the HEAD points at. However, Git also allows you to check out an arbitrary commit that isn't necessarily the tip of any particular branch. The HEAD in such a state is called "detached".

Note that commands that operate on the history of the current branch (e.g. *git commit* to build a new history on top of it) still work while the HEAD is detached. They update the HEAD to point at the tip of the updated history without affecting any branch. Commands that update or inquire information *about* the current branch (e.g. *git branch --set-upstream-to* that sets what remote-tracking branch the current branch integrates with) obviously do not work, as there is no (real) current branch to ask about in this state.

directory

The list you get with "ls" :-)

dirty

A working tree is said to be "dirty" if it contains modifications which have not been committed to the current branch.

evil merge

An evil merge is a merge that introduces changes that do not appear in any parent.

fast-forward

A fast-forward is a special type of merge where you have a revision and you are "merging" another branch's changes that happen to be a descendant of what you have. In such a case, you do not make a new merge commit but instead just update your branch to point at the same revision as the branch you are merging. This will happen frequently on a remote-tracking branch of a remote repository.

fetch

Fetching a branch means to get the branch's head ref from a remote repository, to find out which objects are missing from the local object database, and to get them, too. See also Section G.3.49, “git-fetch(1)”.

file system

Linus Torvalds originally designed Git to be a user space file system, i.e. the infrastructure to hold files and directories. That ensured the efficiency and speed of Git.

Git archive

Synonym for repository (for arch people).

gitfile

A plain file *.git* at the root of a working tree that points at the directory that is the real repository. For proper use see Section G.3.159, “git-worktree(1)” or Section G.3.141, “git-submodule(1)”. For syntax see Section G.4.13, “gitrepository-layout(5)”.

grafts

Grafts enable two otherwise different lines of development to be joined together by recording fake ancestry information for commits. This way you can make Git pretend the set of parents a commit has is different from what was recorded when the commit was created. Configured via the *.git/info/grafts* file.

Note that the grafts mechanism is outdated and can lead to problems transferring objects between repositories; see Section G.3.114, “git-replace(1)” for a more flexible and robust system to do the same thing.

hash

In Git's context, synonym for object name.

head

A named reference to the commit at the tip of a branch. Heads are stored in a file in *$GIT_DIR/refs/heads/* directory, except when using packed refs. (See Section G.3.98, “git-pack-refs(1)”.)

HEAD

The current branch. In more detail: Your working tree is normally derived from the state of the tree referred to by HEAD. HEAD is a reference to one of the heads in your repository, except when using a detached HEAD, in which case it directly references an arbitrary commit.

head ref

A synonym for head.

hook

During the normal execution of several Git commands, call-outs are made to optional scripts that allow a developer to add functionality or checking. Typically, the hooks allow for a command to be pre-verified and potentially aborted, and allow for a post-notification after the operation is done. The hook scripts are found in the *$GIT_DIR/hooks/* directory, and are enabled by simply removing the *.sample* suffix from the filename. In earlier versions of Git you had to make them executable.

index

A collection of files with stat information, whose contents are stored as objects. The index is a stored version of your working tree. Truth be told, it can also contain a second, and even a third version of a working tree, which are used when merging.

index entry

The information regarding a particular file, stored in the index. An index entry can be unmerged, if a merge was started, but not yet finished (i.e. if the index contains multiple versions of that file).

master

The default development branch. Whenever you create a Git repository, a branch named "master" is created, and becomes the active branch. In most cases, this contains the local development, though that is purely by convention and is not required.

merge

As a verb: To bring the contents of another branch (possibly from an external repository) into the current branch. In the case where the merged-in branch is from a different repository, this is done by first fetching the remote branch and then merging the result into the current branch. This combination of fetch and merge operations is called a pull. Merging is performed by an automatic process that identifies changes made since the branches diverged, and then applies all those changes together. In cases where changes conflict, manual intervention may be required to complete the merge.

As a noun: unless it is a fast-forward, a successful merge results in the creation of a new commit representing the result of the merge, and having as parents the tips of the merged branches. This commit is referred to as a "merge commit", or sometimes just a "merge".

object

The unit of storage in Git. It is uniquely identified by the SHA-1 of its contents. Consequently, an object cannot be changed.

object database

Stores a set of "objects", and an individual object is identified by its object name. The objects usually live in *$GIT_DIR/objects/*.

object identifier (oid)

Synonym for object name.

object name

The unique identifier of an object. The object name is usually represented by a 40 character hexadecimal string. Also colloquially called SHA-1.

object type

One of the identifiers "commit", "tree", "tag" or "blob" describing the type of an object.

octopus

To merge more than two branches.

orphan

The act of getting on a branch that does not exist yet (i.e., an unborn branch). After such an operation, the commit first created becomes a commit without a parent, starting a new history.

origin

The default upstream repository. Most projects have at least one upstream project which they track. By default *origin* is used for that purpose. New upstream updates will be fetched into remote-tracking branches named origin/name-of-upstream-branch, which you can see using *git branch -r*.

overlay

Only update and add files to the working directory, but don't delete them, similar to how *cp -R* would update the contents in the destination directory. This is the default mode in a checkout when checking out files from the index or a tree-ish. In contrast, no-overlay mode also deletes tracked files not present in the source, similar to *rsync --delete*.

pack

A set of objects which have been compressed into one file (to save space or to transmit them efficiently).

pack index

The list of identifiers, and other information, of the objects in a pack, to assist in efficiently accessing the contents of a pack.

pathspec

Pattern used to limit paths in Git commands.

Pathspecs are used on the command line of "git ls-files", "git ls-tree", "git add", "git grep", "git diff", "git checkout", and many other commands to limit the scope of operations to some subset of the tree or working tree. See the documentation of each command for whether paths are relative to the current directory or toplevel. The pathspec syntax is as follows:

- any path matches itself
- the pathspec up to the last slash represents a directory prefix. The scope of that pathspec is limited to that subtree.
- the rest of the pathspec is a pattern for the remainder of the pathname. Paths relative to the directory prefix will be matched against that pattern using fnmatch(3); in particular, * and *? can* match directory separators.

For example, Documentation/*.jpg will match all .jpg files in the Documentation subtree, including Documentation/chapter_1/figure_1.jpg.

A pathspec that begins with a colon *:* has special meaning. In the short form, the leading colon *:* is followed by zero or more "magic signature" letters (which optionally is terminated by another colon *:*), and the remainder is the pattern to match against the path. The "magic signature" consists of ASCII symbols that are neither

alphanumeric, glob, regex special characters nor colon. The optional colon that terminates the "magic signature" can be omitted if the pattern begins with a character that does not belong to "magic signature" symbol set and is not a colon.

In the long form, the leading colon *:* is followed by an open parenthesis *(*, a comma-separated list of zero or more "magic words", and a close parentheses *)*, and the remainder is the pattern to match against the path.

A pathspec with only a colon means "there is no pathspec". This form should not be combined with other pathspec.

top

The magic word *top* (magic signature: */*) makes the pattern match from the root of the working tree, even when you are running the command from inside a subdirectory.

literal

Wildcards in the pattern such as * or *?* are treated as literal characters.

icase

Case insensitive match.

glob

Git treats the pattern as a shell glob suitable for consumption by fnmatch(3) with the FNM_PATHNAME flag: wildcards in the pattern will not match a / in the pathname. For example, "Documentation/*.html" matches "Documentation/git.html" but not "Documentation/ppc/ppc.html" or "tools/perf/Documentation/perf.html".

Two consecutive asterisks ("**") in patterns matched against full pathname may have special meaning:

- A leading "**" followed by a slash means match in all directories. For example, "***/foo*" matches file or directory "*foo*" anywhere, the same as pattern "*foo*". "***/foo/bar*" matches file or directory "*bar*" anywhere that is directly under directory "*foo*".
- A trailing "/**" matches everything inside. For example, "*abc/***" matches all files inside directory "abc", relative to the location of the *.gitignore* file, with infinite depth.
- A slash followed by two consecutive asterisks then a slash matches zero or more directories. For example, "*a/**/b*" matches "*a/b*", "*a/x/b*", "*a/x/y/b*" and so on.
- Other consecutive asterisks are considered invalid.

  Glob magic is incompatible with literal magic.

attr

After *attr:* comes a space separated list of "attribute requirements", all of which must be met in order for the path to be considered a match; this is in addition to the usual non-magic pathspec pattern matching. See Section G.4.2, “gitattributes(5)”.

Each of the attribute requirements for the path takes one of these forms:

- "*ATTR*" requires that the attribute *ATTR* be set.
- "*-ATTR*" requires that the attribute *ATTR* be unset.
- "*ATTR=VALUE*" requires that the attribute *ATTR* be set to the string *VALUE*.
- "*!ATTR*" requires that the attribute *ATTR* be unspecified.

  Note that when matching against a tree object, attributes are still obtained from working tree, not from the given tree object.

exclude

After a path matches any non-exclude pathspec, it will be run through all exclude pathspecs (magic signature: *!* or its synonym ^). If it matches, the path is ignored. When there is no non-exclude pathspec, the exclusion is applied to the result set as if invoked without any pathspec.

parent

A commit object contains a (possibly empty) list of the logical predecessor(s) in the line of development, i.e. its parents.

peel

The action of recursively dereferencing a tag object.

pickaxe

The term pickaxe refers to an option to the diffcore routines that help select changes that add or delete a given text string. With the *--pickaxe-all* option, it can be used to view the full changeset that introduced or removed, say, a particular line of text. See Section G.3.44, “git-diff(1)”.

plumbing

Cute name for core Git.

porcelain

Cute name for programs and program suites depending on core Git, presenting a high level access to core Git. Porcelains expose more of a SCM interface than the plumbing.

per-worktree ref

Refs that are per-worktree, rather than global. This is presently only HEAD and any refs that start with *refs/bisect/*, but might later include other unusual refs.

pseudoref

Pseudorefs are a class of files under *$GIT_DIR* which behave like refs for the purposes of rev-parse, but which are treated specially by git. Pseudorefs both have names that are all-caps, and always start with a line consisting of a SHA-1 followed by whitespace. So, HEAD is not a pseudoref, because it is sometimes a symbolic ref. They might optionally contain some additional data. *MERGE_HEAD* and *CHERRY_PICK_HEAD* are examples. Unlike per-worktree refs, these files cannot be symbolic refs, and never have reflogs. They also cannot be updated through the normal ref update machinery. Instead, they are updated by directly writing to the files. However, they can be read as if they were refs, so *git rev-parse MERGE_HEAD* will work.

pull

Pulling a branch means to fetch it and merge it. See also Section G.3.102, “git-pull(1)”.

push

Pushing a branch means to get the branch's head ref from a remote repository, find out if it is an ancestor to the branch's local head ref, and in that case, putting all objects, which are reachable from the local head ref, and which are missing from the remote repository, into the remote object database, and updating the remote head ref. If the remote head is not an ancestor to the local head, the push fails.

reachable

All of the ancestors of a given commit are said to be "reachable" from that commit. More generally, one object is reachable from another if we can reach the one from the other by a chain that follows tags to whatever they tag, commits to their parents or trees, and trees to the trees or blobs that they contain.

reachability bitmaps

Reachability bitmaps store information about the reachability of a selected set of commits in a packfile, or a multi-pack index (MIDX), to speed up object search. The bitmaps are stored in a ".bitmap" file. A repository may have at most one bitmap file in use. The bitmap file may belong to either one pack, or the repository's multi-pack index (if it exists).

rebase

To reapply a series of changes from a branch to a different base, and reset the head of that branch to the result.

ref

A name that begins with *refs/* (e.g. *refs/heads/master*) that points to an object name or another ref (the latter is called a symbolic ref). For convenience, a ref can sometimes be abbreviated when used as an argument to a Git command; see Section G.4.14, “gitrevisions(7)” for details. Refs are stored in the repository.

The ref namespace is hierarchical. Different subhierarchies are used for different purposes (e.g. the *refs/heads/* hierarchy is used to represent local branches).

There are a few special-purpose refs that do not begin with *refs/*. The most notable example is *HEAD*.

reflog

A reflog shows the local "history" of a ref. In other words, it can tell you what the 3rd last revision in *this* repository was, and what was the current state in *this* repository, yesterday 9:14pm. See Section G.3.109, “git-reflog(1)” for details.

refspec

A "refspec" is used by fetch and push to describe the mapping between remote ref and local ref.

remote repository

A repository which is used to track the same project but resides somewhere else. To communicate with remotes, see fetch or push.

remote-tracking branch

A ref that is used to follow changes from another repository. It typically looks like *refs/remotes/foo/bar* (indicating that it tracks a branch named *bar* in a remote named *foo*), and matches the right-hand-side of a configured fetch refspec. A remote-tracking branch should not contain direct modifications or have local commits made to it.

repository

A collection of refs together with an object database containing all objects which are reachable from the refs, possibly accompanied by meta data from one or more porcelains. A repository can share an object database with other repositories via alternates mechanism.

resolve

The action of fixing up manually what a failed automatic merge left behind.

revision

Synonym for commit (the noun).

rewind

To throw away part of the development, i.e. to assign the head to an earlier revision.

SCM

Source code management (tool).

SHA-1

"Secure Hash Algorithm 1"; a cryptographic hash function. In the context of Git used as a synonym for object name.

shallow clone

Mostly a synonym to shallow repository but the phrase makes it more explicit that it was created by running *git clone --depth=...* command.

shallow repository

A shallow repository has an incomplete history some of whose commits have parents cauterized away (in other words, Git is told to pretend that these commits do not have the parents, even though they are recorded in the commit object). This is sometimes useful when you are interested only in the recent history of a project even though the real history recorded in the upstream is much larger. A shallow repository is created by giving the *--depth* option to Section G.3.24, “git-clone(1)”, and its history can be later deepened with Section G.3.49, “git-fetch(1)”.

stash entry

An object used to temporarily store the contents of a dirty working directory and the index for future reuse.

special ref

A ref that has different semantics than normal refs. These refs can be accessed via normal Git commands but may not behave the same as a normal ref in some cases.

The following special refs are known to Git:

- "*FETCH_HEAD*" is written by Section G.3.49, “git-fetch(1)” or Section G.3.102, “git-pull(1)”. It may refer to multiple object IDs. Each object ID is annotated with metadata indicating where it was fetched from and its fetch status.
- "*MERGE_HEAD*" is written by Section G.3.86, “git-merge(1)” when resolving merge conflicts. It contains all commit IDs which are being merged.

submodule

A repository that holds the history of a separate project inside another repository (the latter of which is called superproject).

superproject

A repository that references repositories of other projects in its working tree as submodules. The superproject knows about the names of (but does not hold copies of) commit objects of the contained submodules.

symref

Symbolic reference: instead of containing the SHA-1 id itself, it is of the format *ref: refs/some/thing* and when referenced, it recursively dereferences to this reference. *HEAD* is a prime example of a symref. Symbolic references are manipulated with the Section G.3.143, “git-symbolic-ref(1)” command.

tag

A ref under *refs/tags/* namespace that points to an object of an arbitrary type (typically a tag points to either a tag or a commit object). In contrast to a head, a tag is not updated by the *commit* command. A Git tag has

nothing to do with a Lisp tag (which would be called an object type in Git's context). A tag is most typically used to mark a particular point in the commit ancestry chain.

tag object

An object containing a ref pointing to another object, which can contain a message just like a commit object. It can also contain a (PGP) signature, in which case it is called a "signed tag object".

topic branch

A regular Git branch that is used by a developer to identify a conceptual line of development. Since branches are very easy and inexpensive, it is often desirable to have several small branches that each contain very well defined concepts or small incremental yet related changes.

tree

Either a working tree, or a tree object together with the dependent blob and tree objects (i.e. a stored representation of a working tree).

tree object

An object containing a list of file names and modes along with refs to the associated blob and/or tree objects. A tree is equivalent to a directory.

tree-ish (also treeish)

A tree object or an object that can be recursively dereferenced to a tree object. Dereferencing a commit object yields the tree object corresponding to the revision's top directory. The following are all tree-ishes: a commit-ish, a tree object, a tag object that points to a tree object, a tag object that points to a tag object that points to a tree object, etc.

unborn

The HEAD can point at a branch that does not yet exist and that does not have any commit on it yet, and such a branch is called an unborn branch. The most typical way users encounter an unborn branch is by creating a repository anew without cloning from elsewhere. The HEAD would point at the *main* (or *master*, depending on your configuration) branch that is yet to be born. Also some operations can get you on an unborn branch with their orphan option.

unmerged index

An index which contains unmerged index entries.

unreachable object

An object which is not reachable from a branch, tag, or any other reference.

upstream branch

The default branch that is merged into the branch in question (or the branch in question is rebased onto). It is configured via branch.<name>.remote and branch.<name>.merge. If the upstream branch of *A* is *origin/B* sometimes we say "*A* is tracking *origin/B*".

working tree

The tree of actual checked out files. The working tree normally contains the contents of the HEAD commit's tree, plus any local changes that you have made but not yet committed.

worktree

A repository can have zero (i.e. bare repository) or one or more worktrees attached to it. One "worktree" consists of a "working tree" and repository metadata, most of which are shared among other worktrees of

a single repository, and some of which are maintained separately per worktree (e.g. the index, HEAD and pseudorefs like MERGE_HEAD, per-worktree refs and per-worktree configuration file).

G.1.1.1. Git Quick Reference

This is a quick summary of the major commands; the previous chapters explain how these work in more detail.

1. Creating a new repository

From a tarball:

```
$ tar xzf project.tar.gz
$ cd project
$ git init
Initialized empty Git repository in .git/
$ git add .
$ git commit
```

From a remote repository:

```
$ git clone git://example.com/pub/project.git
$ cd project
```

2. Managing branches

```
$ git branch                # list all local branches in this repo
$ git switch test           # switch working directory to branch "test"
$ git branch new            # create branch "new" starting at current HEAD
$ git branch -d new         # delete branch "new"
```

Instead of basing a new branch on current HEAD (the default), use:

```
$ git branch new test    # branch named "test"
$ git branch new v2.6.15 # tag named v2.6.15
$ git branch new HEAD^   # commit before the most recent
$ git branch new HEAD^^  # commit before that
$ git branch new test~10 # ten commits before tip of branch "test"
```

Create and switch to a new branch at the same time:

```
$ git switch -c new v2.6.15
```

Update and examine branches from the repository you cloned from:

```
$ git fetch            # update
$ git branch -r        # list
  origin/master
  origin/next
  ...
$ git switch -c masterwork origin/master
```

Fetch a branch from a different repository, and give it a new name in your repository:

```
$ git fetch git://example.com/project.git theirbranch:mybranch
$ git fetch git://example.com/project.git v2.6.15:mybranch
```

Keep a list of repositories you work with regularly:

```
$ git remote add example git://example.com/project.git
$ git remote                # list remote repositories
example
```

```
origin
$ git remote show example       # get details
* remote example
  URL: git://example.com/project.git
  Tracked remote branches
    master
    next
    ...
$ git fetch example             # update branches from example
$ git branch -r                 # list all remote branches
```

3. Exploring history

```
$ gitk                      # visualize and browse history
$ git log                   # list all commits
$ git log src/              # ...modifying src/
$ git log v2.6.15..v2.6.16  # ...in v2.6.16, not in v2.6.15
$ git log master..test      # ...in branch test, not in branch master
$ git log test..master      # ...in branch master, but not in test
$ git log test...master     # ...in one branch, not in both
$ git log -S'foo()'         # ...where difference contain "foo()"
$ git log --since="2 weeks ago"
$ git log -p                # show patches as well
$ git show                  # most recent commit
$ git diff v2.6.15..v2.6.16 # diff between two tagged versions
$ git diff v2.6.15..HEAD    # diff with current head
$ git grep "foo()"          # search working directory for "foo()"
$ git grep v2.6.15 "foo()"  # search old tree for "foo()"
$ git show v2.6.15:a.txt    # look at old version of a.txt
```

Search for regressions:

```
$ git bisect start
$ git bisect bad                # current version is bad
$ git bisect good v2.6.13-rc2   # last known good revision
Bisecting: 675 revisions left to test after this
                                # test here, then:
$ git bisect good               # if this revision is good, or
$ git bisect bad                # if this revision is bad.
                                # repeat until done.
```

4. Making changes

Make sure Git knows who to blame:

```
$ cat >>~/.gitconfig <<\EOF
[user]
        name = Your Name Comes Here
        email = you@yourdomain.example.com
EOF
```

Select file contents to include in the next commit, then make the commit:

```
$ git add a.txt    # updated file
$ git add b.txt    # new file
$ git rm c.txt     # old file
$ git commit
```

Or, prepare and create the commit in one step:

```
$ git commit d.txt # use latest content only of d.txt
$ git commit -a    # use latest content of all tracked files
```

5. Merging

```
$ git merge test   # merge branch "test" into the current branch
$ git pull git://example.com/project.git master
                # fetch and merge in remote branch
$ git pull . test  # equivalent to git merge test
```

6. Sharing your changes

Importing or exporting patches:

```
$ git format-patch origin..HEAD # format a patch for each commit
                                # in HEAD but not in origin
$ git am mbox # import patches from the mailbox "mbox"
```

Fetch a branch in a different Git repository, then merge into the current branch:

```
$ git pull git://example.com/project.git theirbranch
```

Store the fetched branch into a local branch before merging into the current branch:

```
$ git pull git://example.com/project.git theirbranch:mybranch
```

After creating commits on a local branch, update the remote branch with your commits:

```
$ git push ssh://example.com/project.git mybranch:theirbranch
```

When remote and local branch are both named "test":

```
$ git push ssh://example.com/project.git test
```

Shortcut version for a frequently used remote repository:

```
$ git remote add example ssh://example.com/project.git
$ git push example test
```

7. Repository maintenance

Check for corruption:

```
$ git fsck
```

Recompress, remove unused cruft:

```
$ git gc
```

G.1.1.2. Notes and todo list for this manual

1. Todo list

This is a work in progress.

The basic requirements:

- It must be readable in order, from beginning to end, by someone intelligent with a basic grasp of the UNIX command line, but without any special knowledge of Git. If necessary, any other prerequisites should be specifically mentioned as they arise.

- Whenever possible, section headings should clearly describe the task they explain how to do, in language that requires no more knowledge than necessary: for example, "importing patches into a project" rather than "the *git am* command"

Think about how to create a clear chapter dependency graph that will allow people to get to important topics without necessarily reading everything in between.

Scan *Documentation/* for other stuff left out; in particular:

- howto's
- some of *technical/*?
- hooks
- list of commands in Section G.3.1, “git(1)”

Scan email archives for other stuff left out

Scan man pages to see if any assume more background than this manual provides.

Add more good examples. Entire sections of just cookbook examples might be a good idea; maybe make an "advanced examples" section a standard end-of-chapter section?

Include cross-references to the glossary, where appropriate.

Add a section on working with other version control systems, including CVS, Subversion, and just imports of series of release tarballs.

Write a chapter on using plumbing and writing scripts.

Alternates, clone -reference, etc.

More on recovery from repository corruption. See: https://lore.kernel.org/git/Pine.LNX.4.64.0702272039540.12485@woody.linux-foundation.org/ https://lore.kernel.org/git/Pine.LNX.4.64.0702141033400.3604@woody.linux-foundation.org/

G.2. Git Tutorial

G.2.1. gittutorial(7)

2024-07-02

Revision History

2024-07-02

NAME

gittutorial - A tutorial introduction to Git

SYNOPSIS

git *

DESCRIPTION

This tutorial explains how to import a new project into Git, make changes to it, and share changes with other developers.

If you are instead primarily interested in using Git to fetch a project, for example, to test the latest version, you may prefer to start with the first two chapters of *The Git User's Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html].

First, note that you can get documentation for a command such as *git log --graph* with:

```
$ man git-log
```

or:

```
$ git help log
```

With the latter, you can use the manual viewer of your choice; see Section G.3.63, “git-help(1)” for more information.

It is a good idea to introduce yourself to Git with your name and public email address before doing any operation. The easiest way to do so is:

```
$ git config --global user.name "Your Name Comes Here"
$ git config --global user.email you@yourdomain.example.com
```

Importing a new project

Assume you have a tarball *project.tar.gz* with your initial work. You can place it under Git revision control as follows.

```
$ tar xzf project.tar.gz
$ cd project
$ git init
```

Git will reply

```
Initialized empty Git repository in .git/
```

You've now initialized the working directory--you may notice a new directory created, named *.git*.

Next, tell Git to take a snapshot of the contents of all files under the current directory (note the .), with *git add*:

```
$ git add .
```

This snapshot is now stored in a temporary staging area which Git calls the "index". You can permanently store the contents of the index in the repository with *git commit*:

```
$ git commit
```

This will prompt you for a commit message. You've now stored the first version of your project in Git.

Making changes

Modify some files, then add their updated contents to the index:

```
$ git add file1 file2 file3
```

You are now ready to commit. You can see what is about to be committed using *git diff* with the *--cached* option:

```
$ git diff --cached
```

(Without *--cached*, *git diff* will show you any changes that you've made but not yet added to the index.) You can also get a brief summary of the situation with *git status*:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

        modified:   file1
        modified:   file2
```

```
        modified:   file3
```

If you need to make any further adjustments, do so now, and then add any newly modified content to the index. Finally, commit your changes with:

```
$ git commit
```

This will again prompt you for a message describing the change, and then record a new version of the project.

Alternatively, instead of running *git add* beforehand, you can use

```
$ git commit -a
```

which will automatically notice any modified (but not new) files, add them to the index, and commit, all in one step.

A note on commit messages: Though not required, it's a good idea to begin the commit message with a single short (no more than 50 characters) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout Git. For example, Section G.3.54, “git-format-patch(1)” turns a commit into email, and it uses the title on the Subject line and the rest of the commit in the body.

Git tracks content not files

Many revision control systems provide an *add* command that tells the system to start tracking changes to a new file. Git's *add* command does something simpler and more powerful: *git add* is used both for new and newly modified files, and in both cases it takes a snapshot of the given files and stages that content in the index, ready for inclusion in the next commit.

Viewing project history

At any point you can view the history of your changes using

```
$ git log
```

If you also want to see complete diffs at each step, use

```
$ git log -p
```

Often the overview of the change is useful to get a feel of each step

```
$ git log --stat --summary
```

Managing branches

A single Git repository can maintain multiple branches of development. To create a new branch named *experimental*, use

```
$ git branch experimental
```

If you now run

```
$ git branch
```

you'll get a list of all existing branches:

```
  experimental
* master
```

The *experimental* branch is the one you just created, and the *master* branch is a default branch that was created for you automatically. The asterisk marks the branch you are currently on; type

```
$ git switch experimental
```

to switch to the *experimental* branch. Now edit a file, commit the change, and switch back to the *master* branch:

```
(edit file)
$ git commit -a
$ git switch master
```

Check that the change you made is no longer visible, since it was made on the *experimental* branch and you're back on the *master* branch.

You can make a different change on the *master* branch:

```
(edit file)
$ git commit -a
```

at this point the two branches have diverged, with different changes made in each. To merge the changes made in *experimental* into *master*, run

```
$ git merge experimental
```

If the changes don't conflict, you're done. If there are conflicts, markers will be left in the problematic files showing the conflict;

```
$ git diff
```

will show this. Once you've edited the files to resolve the conflicts,

```
$ git commit -a
```

will commit the result of the merge. Finally,

```
$ gitk
```

will show a nice graphical representation of the resulting history.

At this point you could delete the *experimental* branch with

```
$ git branch -d experimental
```

This command ensures that the changes in the *experimental* branch are already in the current branch.

If you develop on a branch *crazy-idea*, then regret it, you can always delete the branch with

```
$ git branch -D crazy-idea
```

Branches are cheap and easy, so this is a good way to try something out.

Using Git for collaboration

Suppose that Alice has started a new project with a Git repository in */home/alice/project*, and that Bob, who has a home directory on the same machine, wants to contribute.

Bob begins with:

```
bob$ git clone /home/alice/project myrepo
```

This creates a new directory *myrepo* containing a clone of Alice's repository. The clone is on an equal footing with the original project, possessing its own copy of the original project's history.

Bob then makes some changes and commits them:

```
(edit files)
bob$ git commit -a
(repeat as necessary)
```

When he's ready, he tells Alice to pull changes from the repository at */home/bob/myrepo*. She does this with:

```
alice$ cd /home/alice/project
alice$ git pull /home/bob/myrepo master
```

This merges the changes from Bob's *master* branch into Alice's current branch. If Alice has made her own changes in the meantime, then she may need to manually fix any conflicts.

The *pull* command thus performs two operations: it fetches changes from a remote branch, then merges them into the current branch.

Note that in general, Alice would want her local changes committed before initiating this *pull*. If Bob's work conflicts with what Alice did since their histories forked, Alice will use her working tree and the index to resolve conflicts, and existing local changes will interfere with the conflict resolution process (Git will still perform the fetch but will refuse to merge -- Alice will have to get rid of her local changes in some way and pull again when this happens).

Alice can peek at what Bob did without merging first, using the *fetch* command; this allows Alice to inspect what Bob did, using a special symbol *FETCH_HEAD*, in order to determine if he has anything worth pulling, like this:

```
alice$ git fetch /home/bob/myrepo master
alice$ git log -p HEAD..FETCH_HEAD
```

This operation is safe even if Alice has uncommitted local changes. The range notation *HEAD..FETCH_HEAD* means "show everything that is reachable from the *FETCH_HEAD* but exclude anything that is reachable from *HEAD*". Alice already knows everything that leads to her current state (*HEAD*), and reviews what Bob has in his state (*FETCH_HEAD*) that she has not seen with this command.

If Alice wants to visualize what Bob did since their histories forked she can issue the following command:

```
$ gitk HEAD..FETCH_HEAD
```

This uses the same two-dot range notation we saw earlier with *git log*.

Alice may want to view what both of them did since they forked. She can use three-dot form instead of the two-dot form:

```
$ gitk HEAD...FETCH_HEAD
```

This means "show everything that is reachable from either one, but exclude anything that is reachable from both of them".

Please note that these range notation can be used with both *gitk* and *git log*.

After inspecting what Bob did, if there is nothing urgent, Alice may decide to continue working without pulling from Bob. If Bob's history does have something Alice would immediately need, Alice may choose to stash her work-in-progress first, do a *pull*, and then finally unstash her work-in-progress on top of the resulting history.

When you are working in a small closely knit group, it is not unusual to interact with the same repository over and over again. By defining *remote* repository shorthand, you can make it easier:

```
alice$ git remote add bob /home/bob/myrepo
```

With this, Alice can perform the first part of the *pull* operation alone using the *git fetch* command without merging them with her own branch, using:

```
alice$ git fetch bob
```

Unlike the longhand form, when Alice fetches from Bob using a remote repository shorthand set up with *git remote*, what was fetched is stored in a remote-tracking branch, in this case *bob/master*. So after this:

```
alice$ git log -p master..bob/master
```

shows a list of all the changes that Bob made since he branched from Alice's *master* branch.

After examining those changes, Alice could merge the changes into her *master* branch:

```
alice$ git merge bob/master
```

This *merge* can also be done by *pulling from her own remote-tracking branch*, like this:

```
alice$ git pull . remotes/bob/master
```

Note that git pull always merges into the current branch, regardless of what else is given on the command line.

Later, Bob can update his repo with Alice's latest changes using

```
bob$ git pull
```

Note that he doesn't need to give the path to Alice's repository; when Bob cloned Alice's repository, Git stored the location of her repository in the repository configuration, and that location is used for pulls:

```
bob$ git config --get remote.origin.url
/home/alice/project
```

(The complete configuration created by *git clone* is visible using *git config -l*, and the Section G.3.29, “git-config(1)” man page explains the meaning of each option.)

Git also keeps a pristine copy of Alice's *master* branch under the name *origin/master*:

```
bob$ git branch -r
  origin/master
```

If Bob later decides to work from a different host, he can still perform clones and pulls using the ssh protocol:

```
bob$ git clone alice.org:/home/alice/project myrepo
```

Alternatively, Git has a native protocol, or can use http; see Section G.3.102, “git-pull(1)” for details.

Git can also be used in a CVS-like mode, with a central repository that various users push changes to; see Section G.3.103, “git-push(1)” and Section G.2.4, “gitcvs-migration(7)”.

Exploring history

Git history is represented as a series of interrelated commits. We have already seen that the *git log* command can list those commits. Note that first line of each *git log* entry also gives a name for the commit:

```
$ git log
commit c82a22c39cbc32576f64f5c6b3f24b99ea8149c7
Author: Junio C Hamano <junkio@cox.net>
Date:   Tue May 16 17:18:22 2006 -0700

    merge-base: Clarify the comments on post processing.
```

We can give this name to *git show* to see the details about this commit.

```
$ git show c82a22c39cbc32576f64f5c6b3f24b99ea8149c7
```

But there are other ways to refer to commits. You can use any initial part of the name that is long enough to uniquely identify the commit:

```
$ git show c82a22c39c   # the first few characters of the name are
                        # usually enough
$ git show HEAD         # the tip of the current branch
$ git show experimental # the tip of the "experimental" branch
```

Every commit usually has one "parent" commit which points to the previous state of the project:

```
$ git show HEAD^  # to see the parent of HEAD
$ git show HEAD^^ # to see the grandparent of HEAD
$ git show HEAD~4 # to see the great-great grandparent of HEAD
```

Note that merge commits may have more than one parent:

```
$ git show HEAD^1 # show the first parent of HEAD (same as HEAD^)
$ git show HEAD^2 # show the second parent of HEAD
```

You can also give commits names of your own; after running

```
$ git tag v2.5 1b2e1d63ff
```

you can refer to *1b2e1d63ff* by the name *v2.5*. If you intend to share this name with other people (for example, to identify a release version), you should create a "tag" object, and perhaps sign it; see Section G.3.144, “git-tag(1)” for details.

Any Git command that needs to know a commit can take any of these names. For example:

```
$ git diff v2.5 HEAD     # compare the current HEAD to v2.5
$ git branch stable v2.5 # start a new branch named "stable" based
                         # at v2.5
$ git reset --hard HEAD^ # reset your current branch and working
                         # directory to its state at HEAD^
```

Be careful with that last command: in addition to losing any changes in the working directory, it will also remove all later commits from this branch. If this branch is the only branch containing those commits, they will be lost. Also, don't use *git reset* on a publicly-visible branch that other developers pull from, as it will force needless merges on other developers to clean up the history. If you need to undo changes that you have pushed, use *git revert* instead.

The *git grep* command can search for strings in any version of your project, so

```
$ git grep "hello" v2.5
```

searches for all occurrences of "hello" in *v2.5*.

If you leave out the commit name, *git grep* will search any of the files it manages in your current directory. So

```
$ git grep "hello"
```

is a quick way to search just the files that are tracked by Git.

Many Git commands also take sets of commits, which can be specified in a number of ways. Here are some examples with *git log*:

```
$ git log v2.5..v2.6            # commits between v2.5 and v2.6
$ git log v2.5..                # commits since v2.5
$ git log --since="2 weeks ago" # commits from the last 2 weeks
$ git log v2.5.. Makefile       # commits since v2.5 which modify
                                # Makefile
```

You can also give *git log* a "range" of commits where the first is not necessarily an ancestor of the second; for example, if the tips of the branches *stable* and *master* diverged from a common commit some time ago, then

```
$ git log stable..master
```

will list commits made in the *master* branch but not in the stable branch, while

```
$ git log master..stable
```

will show the list of commits made on the stable branch but not the *master* branch.

The *git log* command has a weakness: it must present commits in a list. When the history has lines of development that diverged and then merged back together, the order in which *git log* presents those commits is meaningless.

Most projects with multiple contributors (such as the Linux kernel, or Git itself) have frequent merges, and *gitk* does a better job of visualizing their history. For example,

```
$ gitk --since="2 weeks ago" drivers/
```

allows you to browse any commits from the last 2 weeks of commits that modified files under the *drivers* directory. (Note: you can adjust gitk's fonts by holding down the control key while pressing "-" or "+".)

Finally, most commands that take filenames will optionally allow you to precede any filename by a commit, to specify a particular version of the file:

```
$ git diff v2.5:Makefile HEAD:Makefile.in
```

You can also use *git show* to see any such file:

```
$ git show v2.5:Makefile
```

Next Steps

This tutorial should be enough to perform basic distributed revision control for your projects. However, to fully understand the depth and power of Git you need to understand two simple ideas on which it is based:

- The object database is the rather elegant system used to store the history of your project--files, directories, and commits.
- The index file is a cache of the state of a directory tree, used to create commits, check out working directories, and hold the various trees involved in a merge.

Part two of this tutorial explains the object database, the index file, and a few other odds and ends that you'll need to make the most of Git. You can find it at Section G.2.2, “gittutorial-2(7)”.

If you don't want to continue with that right away, a few other digressions that may be interesting at this point are:

- Section G.3.54, “git-format-patch(1)”, Section G.3.3, “git-am(1)”: These convert series of git commits into emailed patches, and vice versa, useful for projects such as the Linux kernel which rely heavily on emailed patches.
- Section G.3.8, “git-bisect(1)”: When there is a regression in your project, one way to track down the bug is by searching through the history to find the exact commit that's to blame. *git bisect* can help you perform a binary search for that commit. It is smart enough to perform a close-to-optimal search even in the case of complex non-linear history with lots of merged branches.
- Section G.4.18, “gitworkflows(7)”: Gives an overview of recommended workflows.
- Section G.2.5, “giteveryday(7)”: Everyday Git with 20 Commands Or So.
- Section G.2.4, “gitcvs-migration(7)”: Git for CVS users.

SEE ALSO

Section G.2.2, “gittutorial-2(7)”, Section G.2.4, “gitcvs-migration(7)”, Section G.2.3, “gitcore-tutorial(7)”, Section G.4.19, “gitglossary(7)”, Section G.3.63, “git-help(1)”, Section G.4.18, “gitworkflows(7)”, Section G.2.5, “giteveryday(7)”, *The Git User's Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html]

GIT

Part of the Section G.3.1, “git(1)” suite

G.2.2. gittutorial-2(7)

2024-07-02
Revision History
2024-07-02

NAME

gittutorial-2 - A tutorial introduction to Git: part two

SYNOPSIS

git *

DESCRIPTION

You should work through Section G.2.1, “gittutorial(7)” before reading this tutorial.

The goal of this tutorial is to introduce two fundamental pieces of Git's architecture--the object database and the index file--and to provide the reader with everything necessary to understand the rest of the Git documentation.

The Git object database

Let's start a new project and create a small amount of history:

```
$ mkdir test-project
$ cd test-project
$ git init
Initialized empty Git repository in .git/
$ echo 'hello world' > file.txt
$ git add .
$ git commit -a -m "initial commit"
[master (root-commit) 54196cc] initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 file.txt
$ echo 'hello world!' >file.txt
$ git commit -a -m "add emphasis"
[master c4d59f3] add emphasis
 1 file changed, 1 insertion(+), 1 deletion(-)
```

What are the 7 digits of hex that Git responded to the commit with?

We saw in part one of the tutorial that commits have names like this. It turns out that every object in the Git history is stored under a 40-digit hex name. That name is the SHA-1 hash of the object's contents; among other things, this ensures that Git will never store the same data twice (since identical data is given an identical SHA-1 name), and that the contents of a Git object will never change (since that would change the object's name as well). The 7 char hex strings here are simply the abbreviation of such 40 character long strings. Abbreviations can be used everywhere where the 40 character strings can be used, so long as they are unambiguous.

It is expected that the content of the commit object you created while following the example above generates a different SHA-1 hash than the one shown above because the commit object records the time when it was created and the name of the person performing the commit.

We can ask Git about this particular object with the *cat-file* command. Don't copy the 40 hex digits from this example but use those from your own version. Note that you can shorten it to only a few characters to save yourself typing all 40 hex digits:

```
$ git cat-file -t 54196cc2
commit
$ git cat-file commit 54196cc2
tree 92b8b694ffb1675e5975148e1121810081dbdffe
```

```
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500

initial commit
```

A tree can refer to one or more "blob" objects, each corresponding to a file. In addition, a tree can also refer to other tree objects, thus creating a directory hierarchy. You can examine the contents of any tree using ls-tree (remember that a long enough initial portion of the SHA-1 will also work):

```
$ git ls-tree 92b8b694
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    file.txt
```

Thus we see that this tree has one file in it. The SHA-1 hash is a reference to that file's data:

```
$ git cat-file -t 3b18e512
blob
```

A "blob" is just file data, which we can also examine with cat-file:

```
$ git cat-file blob 3b18e512
hello world
```

Note that this is the old file data; so the object that Git named in its response to the initial tree was a tree with a snapshot of the directory state that was recorded by the first commit.

All of these objects are stored under their SHA-1 names inside the Git directory:

```
$ find .git/objects/
.git/objects/
.git/objects/pack
.git/objects/info
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/92
.git/objects/92/b8b694ffb1675e5975148e1121810081dbdffe
.git/objects/54
.git/objects/54/196cc2703dc165cbd373a65a4dcf22d50ae7f7
.git/objects/a0
.git/objects/a0/423896973644771497bdc03eb99d5281615b51
.git/objects/d0
.git/objects/d0/492b368b66bdabf2ac1fd8c92b39d3db916e59
.git/objects/c4
.git/objects/c4/d59f390b9cfd4318117afde11d601c1085f241
```

and the contents of these files is just the compressed data plus a header identifying their length and their type. The type is either a blob, a tree, a commit, or a tag.

The simplest commit to find is the HEAD commit, which we can find from .git/HEAD:

```
$ cat .git/HEAD
ref: refs/heads/master
```

As you can see, this tells us which branch we're currently on, and it tells us this by naming a file under the .git directory, which itself contains a SHA-1 name referring to a commit object, which we can examine with cat-file:

```
$ cat .git/refs/heads/master
c4d59f390b9cfd4318117afde11d601c1085f241
$ git cat-file -t c4d59f39
commit
$ git cat-file commit c4d59f39
tree d0492b368b66bdabf2ac1fd8c92b39d3db916e59
```

```
parent 54196cc2703dc165cbd373a65a4dcf22d50ae7f7
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143418702 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143418702 -0500

add emphasis
```

The "tree" object here refers to the new state of the tree:

```
$ git ls-tree d0492b36
100644 blob a0423896973644771497bdc03eb99d5281615b51    file.txt
$ git cat-file blob a0423896
hello world!
```

and the "parent" object refers to the previous commit:

```
$ git cat-file commit 54196cc2
tree 92b8b694ffb1675e5975148e1121810081dbdffe
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500

initial commit
```

The tree object is the tree we examined first, and this commit is unusual in that it lacks any parent.

Most commits have only one parent, but it is also common for a commit to have multiple parents. In that case the commit represents a merge, with the parent references pointing to the heads of the merged branches.

Besides blobs, trees, and commits, the only remaining type of object is a "tag", which we won't discuss here; refer to Section G.3.144, “git-tag(1)” for details.

So now we know how Git uses the object database to represent a project's history:

- "commit" objects refer to "tree" objects representing the snapshot of a directory tree at a particular point in the history, and refer to "parent" commits to show how they're connected into the project history.
- "tree" objects represent the state of a single directory, associating directory names to "blob" objects containing file data and "tree" objects containing subdirectory information.
- "blob" objects contain file data without any other structure.
- References to commit objects at the head of each branch are stored in files under .git/refs/heads/.
- The name of the current branch is stored in .git/HEAD.

Note, by the way, that lots of commands take a tree as an argument. But as we can see above, a tree can be referred to in many different ways--by the SHA-1 name for that tree, by the name of a commit that refers to the tree, by the name of a branch whose head refers to that tree, etc.--and most such commands can accept any of these names.

In command synopses, the word "tree-ish" is sometimes used to designate such an argument.

The index file

The primary tool we've been using to create commits is *git-commit -a*, which creates a commit including every change you've made to your working tree. But what if you want to commit changes only to certain files? Or only certain changes to certain files?

If we look at the way commits are created under the cover, we'll see that there are more flexible ways creating commits.

Continuing with our test-project, let's modify file.txt again:

```
$ echo "hello world, again" >>file.txt
```

but this time instead of immediately making the commit, let's take an intermediate step, and ask for diffs along the way to keep track of what's happening:

```
$ git diff
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
 hello world!
+hello world, again
$ git add file.txt
$ git diff
```

The last diff is empty, but no new commits have been made, and the head still doesn't contain the new line:

```
$ git diff HEAD
diff --git a/file.txt b/file.txt
index a042389..513feba 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
 hello world!
+hello world, again
```

So *git diff* is comparing against something other than the head. The thing that it's comparing against is actually the index file, which is stored in .git/index in a binary format, but whose contents we can examine with ls-files:

```
$ git ls-files --stage
100644 513feba2e53ebbd2532419ded848ba19de88ba00 0       file.txt
$ git cat-file -t 513feba2
blob
$ git cat-file blob 513feba2
hello world!
hello world, again
```

So what our *git add* did was store a new blob and then put a reference to it in the index file. If we modify the file again, we'll see that the new modifications are reflected in the *git diff* output:

```
$ echo 'again?' >>file.txt
$ git diff
index 513feba..ba3da7b 100644
--- a/file.txt
+++ b/file.txt
@@ -1,2 +1,3 @@
 hello world!
 hello world, again
+again?
```

With the right arguments, *git diff* can also show us the difference between the working directory and the last commit, or between the index and the last commit:

```
$ git diff HEAD
diff --git a/file.txt b/file.txt
index a042389..ba3da7b 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,3 @@
 hello world!
+hello world, again
+again?
$ git diff --cached
```

```
diff --git a/file.txt b/file.txt
index a042389..513feba 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
 hello world!
+hello world, again
```

At any time, we can create a new commit using *git commit* (without the "-a" option), and verify that the state committed only includes the changes stored in the index file, not the additional change that is still only in our working tree:

```
$ git commit -m "repeat"
$ git diff HEAD
diff --git a/file.txt b/file.txt
index 513feba..ba3da7b 100644
--- a/file.txt
+++ b/file.txt
@@ -1,2 +1,3 @@
 hello world!
 hello world, again
+again?
```

So by default *git commit* uses the index to create the commit, not the working tree; the "-a" option to commit tells it to first update the index with all changes in the working tree.

Finally, it's worth looking at the effect of *git add* on the index file:

```
$ echo "goodbye, world" >closing.txt
$ git add closing.txt
```

The effect of the *git add* was to add one entry to the index file:

```
$ git ls-files --stage
100644 8b9743b20d4b15be3955fc8d5cd2b09cd2336138 0       closing.txt
100644 513feba2e53ebbd2532419ded848ba19de88ba00 0       file.txt
```

And, as you can see with cat-file, this new entry refers to the current contents of the file:

```
$ git cat-file blob 8b9743b2
goodbye, world
```

The "status" command is a useful way to get a quick summary of the situation:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

        new file:   closing.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)

        modified:   file.txt
```

Since the current state of closing.txt is cached in the index file, it is listed as "Changes to be committed". Since file.txt has changes in the working directory that aren't reflected in the index, it is marked "changed but not updated". At this point, running "git commit" would create a commit that added closing.txt (with its new contents), but that didn't modify file.txt.

Also, note that a bare *git diff* shows the changes to file.txt, but not the addition of closing.txt, because the version of closing.txt in the index file is identical to the one in the working directory.

In addition to being the staging area for new commits, the index file is also populated from the object database when checking out a branch, and is used to hold the trees involved in a merge operation. See Section G.2.3, “gitcore-tutorial(7)” and the relevant man pages for details.

What next?

At this point you should know everything necessary to read the man pages for any of the git commands; one good place to start would be with the commands mentioned in Section G.2.5, “giteveryday(7)”. You should be able to find any unknown jargon in Section G.4.19, “gitglossary(7)”.

The *Git User's Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html] provides a more comprehensive introduction to Git.

Section G.2.4, “gitcvs-migration(7)” explains how to import a CVS repository into Git, and shows how to use Git in a CVS-like way.

For some interesting examples of Git use, see the *howtos* [https://www.kernel.org/pub/software/scm/git/docs/how-to-index.html].

For Git developers, Section G.2.3, “gitcore-tutorial(7)” goes into detail on the lower-level Git mechanisms involved in, for example, creating a new commit.

SEE ALSO

Section G.2.1, “gittutorial(7)”, Section G.2.4, “gitcvs-migration(7)”, Section G.2.3, “gitcore-tutorial(7)”, Section G.4.19, “gitglossary(7)”, Section G.3.63, “git-help(1)”, Section G.2.5, “giteveryday(7)”, *The Git User's Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html]

GIT

Part of the Section G.3.1, “git(1)” suite

G.2.3. gitcore-tutorial(7)

2024-07-02

Revision History

2024-07-02

NAME

gitcore-tutorial - A Git core tutorial for developers

SYNOPSIS

git *

DESCRIPTION

This tutorial explains how to use the "core" Git commands to set up and work with a Git repository.

If you just need to use Git as a revision control system you may prefer to start with "A Tutorial Introduction to Git" (Section G.2.1, “gittutorial(7)”) or *the Git User Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html].

However, an understanding of these low-level tools can be helpful if you want to understand Git's internals.

The core Git is often called "plumbing", with the prettier user interfaces on top of it called "porcelain". You may not want to use the plumbing directly very often, but it can be good to know what the plumbing does when the porcelain isn't flushing.

Back when this document was originally written, many porcelain commands were shell scripts. For simplicity, it still uses them as examples to illustrate how plumbing is fit together to form the porcelain commands. The source tree includes some of these scripts in contrib/examples/ for reference. Although these are not implemented as shell scripts anymore, the description of what the plumbing layer commands do is still valid.

> **Note**
>
> Deeper technical details are often marked as Notes, which you can skip on your first reading.

Creating a Git repository

Creating a new Git repository couldn't be easier: all Git repositories start out empty, and the only thing you need to do is find yourself a subdirectory that you want to use as a working tree - either an empty one for a totally new project, or an existing working tree that you want to import into Git.

For our first example, we're going to start a totally new repository from scratch, with no pre-existing files, and we'll call it *git-tutorial*. To start up, create a subdirectory for it, change into that subdirectory, and initialize the Git infrastructure with *git init*:

```
$ mkdir git-tutorial
$ cd git-tutorial
$ git init
```

to which Git will reply

```
Initialized empty Git repository in .git/
```

which is just Git's way of saying that you haven't been doing anything strange, and that it will have created a local *.git* directory setup for your new project. You will now have a *.git* directory, and you can inspect that with *ls*. For your new empty project, it should show you three entries, among other things:

- a file called *HEAD*, that has *ref: refs/heads/master* in it. This is similar to a symbolic link and points at *refs/heads/master* relative to the *HEAD* file.

  Don't worry about the fact that the file that the *HEAD* link points to doesn't even exist yet -- you haven't created the commit that will start your *HEAD* development branch yet.

- a subdirectory called *objects*, which will contain all the objects of your project. You should never have any real reason to look at the objects directly, but you might want to know that these objects are what contains all the real *data* in your repository.

- a subdirectory called *refs*, which contains references to objects.

In particular, the *refs* subdirectory will contain two other subdirectories, named *heads* and *tags* respectively. They do exactly what their names imply: they contain references to any number of different *heads* of development (aka *branches*), and to any *tags* that you have created to name specific versions in your repository.

One note: the special *master* head is the default branch, which is why the *.git/HEAD* file was created points to it even if it doesn't yet exist. Basically, the *HEAD* link is supposed to always point to the branch you are working on right now, and you always start out expecting to work on the *master* branch.

However, this is only a convention, and you can name your branches anything you want, and don't have to ever even *have* a *master* branch. A number of the Git tools will assume that *.git/HEAD* is valid, though.

> **Note**
>
> An *object* is identified by its 160-bit SHA-1 hash, aka *object name*, and a reference to an object is always the 40-byte hex representation of that SHA-1 name. The files in the *refs* subdirectory are expected to contain these hex references (usually with a final *\n* at the end), and you should thus

expect to see a number of 41-byte files containing these references in these *refs* subdirectories when you actually start populating your tree.

Note

An advanced user may want to take a look at Section G.4.13, “gitrepository-layout(5)” after finishing this tutorial.

You have now created your first Git repository. Of course, since it's empty, that's not very useful, so let's start populating it with data.

Populating a Git repository

We'll keep this simple and stupid, so we'll start off with populating a few trivial files just to get a feel for it.

Start off with just creating any random files that you want to maintain in your Git repository. We'll start off with a few bad examples, just to get a feel for how this works:

```
$ echo "Hello World" >hello
$ echo "Silly example" >example
```

you have now created two files in your working tree (aka *working directory*), but to actually check in your hard work, you will have to go through two steps:

- fill in the *index* file (aka *cache*) with the information about your working tree state.
- commit that index file as an object.

The first step is trivial: when you want to tell Git about any changes to your working tree, you use the *git update-index* program. That program normally just takes a list of filenames you want to update, but to avoid trivial mistakes, it refuses to add new entries to the index (or remove existing ones) unless you explicitly tell it that you're adding a new entry with the *--add* flag (or removing an entry with the *--remove*) flag.

So to populate the index with the two files you just created, you can do

```
$ git update-index --add hello example
```

and you have now told Git to track those two files.

In fact, as you did that, if you now look into your object directory, you'll notice that Git will have added two new objects to the object database. If you did exactly the steps above, you should now be able to do

```
$ ls .git/objects/??/*
```

and see two files:

```
.git/objects/55/7db03de997c86a4a028e1ebd3a1ceb225be238
.git/objects/f2/4c74a2e500f5ee1332c86b94199f52b1d1d962
```

which correspond with the objects with names of *557db*... and *f24c7*... respectively.

If you want to, you can use *git cat-file* to look at those objects, but you'll have to use the object name, not the filename of the object:

```
$ git cat-file -t 557db03de997c86a4a028e1ebd3a1ceb225be238
```

where the *-t* tells *git cat-file* to tell you what the "type" of the object is. Git will tell you that you have a "blob" object (i.e., just a regular file), and you can see the contents with

```
$ git cat-file blob 557db03
```

which will print out "Hello World". The object *557db03* is nothing more than the contents of your file *hello*.

> **Note**
>
> Don't confuse that object with the file *hello* itself. The object is literally just those specific **contents** of the file, and however much you later change the contents in file *hello*, the object we just looked at will never change. Objects are immutable.

> **Note**
>
> The second example demonstrates that you can abbreviate the object name to only the first several hexadecimal digits in most places.

Anyway, as we mentioned previously, you normally never actually take a look at the objects themselves, and typing long 40-character hex names is not something you'd normally want to do. The above digression was just to show that *git update-index* did something magical, and actually saved away the contents of your files into the Git object database.

Updating the index did something else too: it created a *.git/index* file. This is the index that describes your current working tree, and something you should be very aware of. Again, you normally never worry about the index file itself, but you should be aware of the fact that you have not actually really "checked in" your files into Git so far, you've only **told** Git about them.

However, since Git knows about them, you can now start using some of the most basic Git commands to manipulate the files or look at their status.

In particular, let's not even check in the two files into Git yet, we'll start off by adding another line to *hello* first:

```
$ echo "It's a new day for git" >>hello
```

and you can now, since you told Git about the previous state of *hello*, ask Git what has changed in the tree compared to your old index, using the *git diff-files* command:

```
$ git diff-files
```

Oops. That wasn't very readable. It just spit out its own internal version of a *diff*, but that internal version really just tells you that it has noticed that "hello" has been modified, and that the old object contents it had have been replaced with something else.

To make it readable, we can tell *git diff-files* to output the differences as a patch, using the *-p* flag:

```
$ git diff-files -p
diff --git a/hello b/hello
index 557db03..263414f 100644
--- a/hello
+++ b/hello
@@ -1 +1,2 @@
 Hello World
+It's a new day for git
```

i.e. the diff of the change we caused by adding another line to *hello*.

In other words, *git diff-files* always shows us the difference between what is recorded in the index, and what is currently in the working tree. That's very useful.

A common shorthand for *git diff-files -p* is to just write *git diff*, which will do the same thing.

```
$ git diff
diff --git a/hello b/hello
```

```
index 557db03..263414f 100644
--- a/hello
+++ b/hello
@@ -1 +1,2 @@
 Hello World
+It's a new day for git
```

Committing Git state

Now, we want to go to the next stage in Git, which is to take the files that Git knows about in the index, and commit them as a real tree. We do that in two phases: creating a *tree* object, and committing that *tree* object as a *commit* object together with an explanation of what the tree was all about, along with information of how we came to that state.

Creating a tree object is trivial, and is done with *git write-tree*. There are no options or other input: *git write-tree* will take the current index state, and write an object that describes that whole index. In other words, we're now tying together all the different filenames with their contents (and their permissions), and we're creating the equivalent of a Git "directory" object:

```
$ git write-tree
```

and this will just output the name of the resulting tree, in this case (if you have done exactly as I've described) it should be

```
8988da15d077d4829fc51d8544c097def6644dbb
```

which is another incomprehensible object name. Again, if you want to, you can use *git cat-file -t 8988d...* to see that this time the object is not a "blob" object, but a "tree" object (you can also use *git cat-file* to actually output the raw object contents, but you'll see mainly a binary mess, so that's less interesting).

However -- normally you'd never use *git write-tree* on its own, because normally you always commit a tree into a commit object using the *git commit-tree* command. In fact, it's easier to not actually use *git write-tree* on its own at all, but to just pass its result in as an argument to *git commit-tree*.

git commit-tree normally takes several arguments -- it wants to know what the *parent* of a commit was, but since this is the first commit ever in this new repository, and it has no parents, we only need to pass in the object name of the tree. However, *git commit-tree* also wants to get a commit message on its standard input, and it will write out the resulting object name for the commit to its standard output.

And this is where we create the *.git/refs/heads/master* file which is pointed at by *HEAD*. This file is supposed to contain the reference to the top-of-tree of the master branch, and since that's exactly what *git commit-tree* spits out, we can do this all with a sequence of simple shell commands:

```
$ tree=$(git write-tree)
$ commit=$(echo 'Initial commit' | git commit-tree $tree)
$ git update-ref HEAD $commit
```

In this case this creates a totally new commit that is not related to anything else. Normally you do this only **once** for a project ever, and all later commits will be parented on top of an earlier commit.

Again, normally you'd never actually do this by hand. There is a helpful script called *git commit* that will do all of this for you. So you could have just written *git commit* instead, and it would have done the above magic scripting for you.

Making a change

Remember how we did the *git update-index* on file *hello* and then we changed *hello* afterward, and could compare the new state of *hello* with the state we saved in the index file?

Further, remember how I said that *git write-tree* writes the contents of the **index** file to the tree, and thus what we just committed was in fact the **original** contents of the file *hello*, not the new ones. We did that on purpose, to

show the difference between the index state, and the state in the working tree, and how they don't have to match, even when we commit things.

As before, if we do *git diff-files -p* in our git-tutorial project, we'll still see the same difference we saw last time: the index file hasn't changed by the act of committing anything. However, now that we have committed something, we can also learn to use a new command: *git diff-index*.

Unlike *git diff-files*, which showed the difference between the index file and the working tree, *git diff-index* shows the differences between a committed **tree** and either the index file or the working tree. In other words, *git diff-index* wants a tree to be diffed against, and before we did the commit, we couldn't do that, because we didn't have anything to diff against.

But now we can do

```
$ git diff-index -p HEAD
```

(where *-p* has the same meaning as it did in *git diff-files*), and it will show us the same difference, but for a totally different reason. Now we're comparing the working tree not against the index file, but against the tree we just wrote. It just so happens that those two are obviously the same, so we get the same result.

Again, because this is a common operation, you can also just shorthand it with

```
$ git diff HEAD
```

which ends up doing the above for you.

In other words, *git diff-index* normally compares a tree against the working tree, but when given the *--cached* flag, it is told to instead compare against just the index cache contents, and ignore the current working tree state entirely. Since we just wrote the index file to HEAD, doing *git diff-index --cached -p HEAD* should thus return an empty set of differences, and that's exactly what it does.

> **Note**
>
> *git diff-index* really always uses the index for its comparisons, and saying that it compares a tree against the working tree is thus not strictly accurate. In particular, the list of files to compare (the "meta-data") **always** comes from the index file, regardless of whether the *--cached* flag is used or not. The *--cached* flag really only determines whether the file **contents** to be compared come from the working tree or not.
>
> This is not hard to understand, as soon as you realize that Git simply never knows (or cares) about files that it is not told about explicitly. Git will never go **looking** for files to compare, it expects you to tell it what the files are, and that's what the index is there for.

However, our next step is to commit the **change** we did, and again, to understand what's going on, keep in mind the difference between "working tree contents", "index file" and "committed tree". We have changes in the working tree that we want to commit, and we always have to work through the index file, so the first thing we need to do is to update the index cache:

```
$ git update-index hello
```

(note how we didn't need the *--add* flag this time, since Git knew about the file already).

Note what happens to the different *git diff-** versions here. After we've updated *hello* in the index, *git diff-files -p* now shows no differences, but *git diff-index -p HEAD* still **does** show that the current state is different from the state we committed. In fact, now *git diff-index* shows the same difference whether we use the *--cached* flag or not, since now the index is coherent with the working tree.

Now, since we've updated *hello* in the index, we can commit the new version. We could do it by writing the tree by hand again, and committing the tree (this time we'd have to use the *-p HEAD* flag to tell commit that the HEAD

was the **parent** of the new commit, and that this wasn't an initial commit any more), but you've done that once already, so let's just use the helpful script this time:

```
$ git commit
```

which starts an editor for you to write the commit message and tells you a bit about what you have done.

Write whatever message you want, and all the lines that start with # will be pruned out, and the rest will be used as the commit message for the change. If you decide you don't want to commit anything after all at this point (you can continue to edit things and update the index), you can just leave an empty message. Otherwise *git commit* will commit the change for you.

You've now made your first real Git commit. And if you're interested in looking at what *git commit* really does, feel free to investigate: it's a few very simple shell scripts to generate the helpful (?) commit message headers, and a few one-liners that actually do the commit itself (*git commit*).

Inspecting Changes

While creating changes is useful, it's even more useful if you can tell later what changed. The most useful command for this is another of the *diff* family, namely *git diff-tree*.

git diff-tree can be given two arbitrary trees, and it will tell you the differences between them. Perhaps even more commonly, though, you can give it just a single commit object, and it will figure out the parent of that commit itself, and show the difference directly. Thus, to get the same diff that we've already seen several times, we can now do

```
$ git diff-tree -p HEAD
```

(again, *-p* means to show the difference as a human-readable patch), and it will show what the last commit (in *HEAD*) actually changed.

Note

Here is an ASCII art by Jon Loeliger that illustrates how various *diff-** commands compare things.

```
             diff-tree
              +----+
              |    |
              |    |
              V    V
           +-----------+
           | Object DB |
           |  Backing  |
           |   Store   |
           +-----------+
             ^    ^
             |    |
             |    |  diff-index --cached
             |    |
diff-index   |    V
             |  +-----------+
             |  |   Index   |
             |  |  "cache"  |
             |  +-----------+
             |    ^
             |    |
             |    |  diff-files
             |    |
             V    V
           +-----------+
```

```
|  Working  |
| Directory |
+-----------+
```

More interestingly, you can also give *git diff-tree* the *--pretty* flag, which tells it to also show the commit message and author and date of the commit, and you can tell it to show a whole series of diffs. Alternatively, you can tell it to be "silent", and not show the diffs at all, but just show the actual commit message.

In fact, together with the *git rev-list* program (which generates a list of revisions), *git diff-tree* ends up being a veritable fount of changes. You can emulate *git log*, *git log -p*, etc. with a trivial script that pipes the output of *git rev-list* to *git diff-tree --stdin*, which was exactly how early versions of *git log* were implemented.

Tagging a version

In Git, there are two kinds of tags, a "light" one, and an "annotated tag".

A "light" tag is technically nothing more than a branch, except we put it in the *.git/refs/tags/* subdirectory instead of calling it a *head*. So the simplest form of tag involves nothing more than

```
$ git tag my-first-tag
```

which just writes the current *HEAD* into the *.git/refs/tags/my-first-tag* file, after which point you can then use this symbolic name for that particular state. You can, for example, do

```
$ git diff my-first-tag
```

to diff your current state against that tag which at this point will obviously be an empty diff, but if you continue to develop and commit stuff, you can use your tag as an "anchor-point" to see what has changed since you tagged it.

An "annotated tag" is actually a real Git object, and contains not only a pointer to the state you want to tag, but also a small tag name and message, along with optionally a PGP signature that says that yes, you really did that tag. You create these annotated tags with either the *-a* or *-s* flag to *git tag*:

```
$ git tag -s <tagname>
```

which will sign the current *HEAD* (but you can also give it another argument that specifies the thing to tag, e.g., you could have tagged the current *mybranch* point by using *git tag <tagname> mybranch*).

You normally only do signed tags for major releases or things like that, while the light-weight tags are useful for any marking you want to do -- any time you decide that you want to remember a certain point, just create a private tag for it, and you have a nice symbolic name for the state at that point.

Copying repositories

Git repositories are normally totally self-sufficient and relocatable. Unlike CVS, for example, there is no separate notion of "repository" and "working tree". A Git repository normally **is** the working tree, with the local Git information hidden in the *.git* subdirectory. There is nothing else. What you see is what you got.

> **Note**
>
> You can tell Git to split the Git internal information from the directory that it tracks, but we'll ignore that for now: it's not how normal projects work, and it's really only meant for special uses. So the mental model of "the Git information is always tied directly to the working tree that it describes" may not be technically 100% accurate, but it's a good model for all normal use.

This has two implications:

- if you grow bored with the tutorial repository you created (or you've made a mistake and want to start all over), you can just do simple

```
$ rm -rf git-tutorial
```

and it will be gone. There's no external repository, and there's no history outside the project you created.

- if you want to move or duplicate a Git repository, you can do so. There is *git clone* command, but if all you want to do is just to create a copy of your repository (with all the full history that went along with it), you can do so with a regular *cp -a git-tutorial new-git-tutorial*.

  Note that when you've moved or copied a Git repository, your Git index file (which caches various information, notably some of the "stat" information for the files involved) will likely need to be refreshed. So after you do a *cp -a* to create a new copy, you'll want to do

  ```
  $ git update-index --refresh
  ```

  in the new repository to make sure that the index file is up to date.

Note that the second point is true even across machines. You can duplicate a remote Git repository with **any** regular copy mechanism, be it *scp*, *rsync* or *wget*.

When copying a remote repository, you'll want to at a minimum update the index cache when you do this, and especially with other peoples' repositories you often want to make sure that the index cache is in some known state (you don't know **what** they've done and not yet checked in), so usually you'll precede the *git update-index* with a

```
$ git read-tree --reset HEAD
$ git update-index --refresh
```

which will force a total index re-build from the tree pointed to by *HEAD*. It resets the index contents to *HEAD*, and then the *git update-index* makes sure to match up all index entries with the checked-out files. If the original repository had uncommitted changes in its working tree, *git update-index --refresh* notices them and tells you they need to be updated.

The above can also be written as simply

```
$ git reset
```

and in fact a lot of the common Git command combinations can be scripted with the *git xyz* interfaces. You can learn things by just looking at what the various git scripts do. For example, *git reset* used to be the above two lines implemented in *git reset*, but some things like *git status* and *git commit* are slightly more complex scripts around the basic Git commands.

Many (most?) public remote repositories will not contain any of the checked out files or even an index file, and will **only** contain the actual core Git files. Such a repository usually doesn't even have the *.git* subdirectory, but has all the Git files directly in the repository.

To create your own local live copy of such a "raw" Git repository, you'd first create your own subdirectory for the project, and then copy the raw repository contents into the *.git* directory. For example, to create your own copy of the Git repository, you'd do the following

```
$ mkdir my-git
$ cd my-git
$ rsync -rL rsync://rsync.kernel.org/pub/scm/git/git.git/ .git
```

followed by

```
$ git read-tree HEAD
```

to populate the index. However, now you have populated the index, and you have all the Git internal files, but you will notice that you don't actually have any of the working tree files to work on. To get those, you'd check them out with

```
$ git checkout-index -u -a
```

where the *-u* flag means that you want the checkout to keep the index up to date (so that you don't have to refresh it afterward), and the *-a* flag means "check out all files" (if you have a stale copy or an older version of a checked out tree you may also need to add the *-f* flag first, to tell *git checkout-index* to **force** overwriting of any old files).

Again, this can all be simplified with

```
$ git clone git://git.kernel.org/pub/scm/git/git.git/ my-git
$ cd my-git
$ git checkout
```

which will end up doing all of the above for you.

You have now successfully copied somebody else's (mine) remote repository, and checked it out.

Creating a new branch

Branches in Git are really nothing more than pointers into the Git object database from within the *.git/refs/* subdirectory, and as we already discussed, the *HEAD* branch is nothing but a symlink to one of these object pointers.

You can at any time create a new branch by just picking an arbitrary point in the project history, and just writing the SHA-1 name of that object into a file under *.git/refs/heads/*. You can use any filename you want (and indeed, subdirectories), but the convention is that the "normal" branch is called *master*. That's just a convention, though, and nothing enforces it.

To show that as an example, let's go back to the git-tutorial repository we used earlier, and create a branch in it. You do that by simply just saying that you want to check out a new branch:

```
$ git switch -c mybranch
```

will create a new branch based at the current *HEAD* position, and switch to it.

> **Note**
>
> If you make the decision to start your new branch at some other point in the history than the current *HEAD*, you can do so by just telling *git switch* what the base of the checkout would be. In other words, if you have an earlier tag or branch, you'd just do
>
> ```
> $ git switch -c mybranch earlier-commit
> ```
>
> and it would create the new branch *mybranch* at the earlier commit, and check out the state at that time.

You can always just jump back to your original *master* branch by doing

```
$ git switch master
```

(or any other branch-name, for that matter) and if you forget which branch you happen to be on, a simple

```
$ cat .git/HEAD
```

will tell you where it's pointing. To get the list of branches you have, you can say

```
$ git branch
```

which used to be nothing more than a simple script around *ls .git/refs/heads*. There will be an asterisk in front of the branch you are currently on.

Sometimes you may wish to create a new branch *without* actually checking it out and switching to it. If so, just use the command

```
$ git branch <branchname> [startingpoint]
```

which will simply *create* the branch, but will not do anything further. You can then later -- once you decide that you want to actually develop on that branch -- switch to that branch with a regular *git switch* with the branchname as the argument.

Merging two branches

One of the ideas of having a branch is that you do some (possibly experimental) work in it, and eventually merge it back to the main branch. So assuming you created the above *mybranch* that started out being the same as the original *master* branch, let's make sure we're in that branch, and do some work there.

```
$ git switch mybranch
$ echo "Work, work, work" >>hello
$ git commit -m "Some work." -i hello
```

Here, we just added another line to *hello*, and we used a shorthand for doing both *git update-index hello* and *git commit* by just giving the filename directly to *git commit*, with an *-i* flag (it tells Git to *include* that file in addition to what you have done to the index file so far when making the commit). The *-m* flag is to give the commit log message from the command line.

Now, to make it a bit more interesting, let's assume that somebody else does some work in the original branch, and simulate that by going back to the master branch, and editing the same file differently there:

```
$ git switch master
```

Here, take a moment to look at the contents of *hello*, and notice how they don't contain the work we just did in *mybranch* -- because that work hasn't happened in the *master* branch at all. Then do

```
$ echo "Play, play, play" >>hello
$ echo "Lots of fun" >>example
$ git commit -m "Some fun." -i hello example
```

since the master branch is obviously in a much better mood.

Now, you've got two branches, and you decide that you want to merge the work done. Before we do that, let's introduce a cool graphical tool that helps you view what's going on:

```
$ gitk --all
```

will show you graphically both of your branches (that's what the *--all* means: normally it will just show you your current *HEAD*) and their histories. You can also see exactly how they came to be from a common source.

Anyway, let's exit *gitk* (*^Q* or the File menu), and decide that we want to merge the work we did on the *mybranch* branch into the *master* branch (which is currently our *HEAD* too). To do that, there's a nice script called *git merge*, which wants to know which branches you want to resolve and what the merge is all about:

```
$ git merge -m "Merge work in mybranch" mybranch
```

where the first argument is going to be used as the commit message if the merge can be resolved automatically.

Now, in this case we've intentionally created a situation where the merge will need to be fixed up by hand, though, so Git will do as much of it as it can automatically (which in this case is just merge the *example* file, which had no differences in the *mybranch* branch), and say:

```
Auto-merging hello
CONFLICT (content): Merge conflict in hello
Automatic merge failed; fix conflicts and then commit the result.
```

It tells you that it did an "Automatic merge", which failed due to conflicts in *hello*.

Not to worry. It left the (trivial) conflict in *hello* in the same form you should already be well used to if you've ever used CVS, so let's just open *hello* in our editor (whatever that may be), and fix it up somehow. I'd suggest just making it so that *hello* contains all four lines:

```
Hello World
It's a new day for git
Play, play, play
Work, work, work
```

and once you're happy with your manual merge, just do a

```
$ git commit -i hello
```

which will very loudly warn you that you're now committing a merge (which is correct, so never mind), and you can write a small merge message about your adventures in *git merge*-land.

After you're done, start up *gitk --all* to see graphically what the history looks like. Notice that *mybranch* still exists, and you can switch to it, and continue to work with it if you want to. The *mybranch* branch will not contain the merge, but next time you merge it from the *master* branch, Git will know how you merged it, so you'll not have to do *that* merge again.

Another useful tool, especially if you do not always work in X-Window environment, is *git show-branch*.

```
$ git show-branch --topo-order --more=1 master mybranch
* [master] Merge work in mybranch
 ! [mybranch] Some work.
--
-  [master] Merge work in mybranch
*+ [mybranch] Some work.
*  [master^] Some fun.
```

The first two lines indicate that it is showing the two branches with the titles of their top-of-the-tree commits, you are currently on *master* branch (notice the asterisk * character), and the first column for the later output lines is used to show commits contained in the *master* branch, and the second column for the *mybranch* branch. Three commits are shown along with their titles. All of them have non blank characters in the first column (* shows an ordinary commit on the current branch, - is a merge commit), which means they are now part of the *master* branch. Only the "Some work" commit has the plus + character in the second column, because *mybranch* has not been merged to incorporate these commits from the master branch. The string inside brackets before the commit log message is a short name you can use to name the commit. In the above example, *master* and *mybranch* are branch heads. *master^* is the first parent of *master* branch head. Please see Section G.4.14, “gitrevisions(7)” if you want to see more complex cases.

> **Note**
>
> Without the *--more=1* option, *git show-branch* would not output the *[master^]* commit, as *[mybranch]* commit is a common ancestor of both *master* and *mybranch* tips. Please see Section G.3.131, “git-show-branch(1)” for details.

> **Note**
>
> If there were more commits on the *master* branch after the merge, the merge commit itself would not be shown by *git show-branch* by default. You would need to provide *--sparse* option to make the merge commit visible in this case.

Now, let's pretend you are the one who did all the work in *mybranch*, and the fruit of your hard work has finally been merged to the *master* branch. Let's go back to *mybranch*, and run *git merge* to get the "upstream changes" back to your branch.

```
$ git switch mybranch
$ git merge -m "Merge upstream changes." master
```

This outputs something like this (the actual commit object names would be different)

```
Updating from ae3a2da... to a80b4aa....
Fast-forward (no commit created; -m option ignored)
 example | 1 +
 hello   | 1 +
 2 files changed, 2 insertions(+)
```

Because your branch did not contain anything more than what had already been merged into the *master* branch, the merge operation did not actually do a merge. Instead, it just updated the top of the tree of your branch to that of the *master* branch. This is often called *fast-forward* merge.

You can run *gitk --all* again to see how the commit ancestry looks like, or run *show-branch*, which tells you this.

```
$ git show-branch master mybranch
! [master] Merge work in mybranch
 * [mybranch] Merge work in mybranch
--
-- [master] Merge work in mybranch
```

Merging external work

It's usually much more common that you merge with somebody else than merging with your own branches, so it's worth pointing out that Git makes that very easy too, and in fact, it's not that different from doing a *git merge*. In fact, a remote merge ends up being nothing more than "fetch the work from a remote repository into a temporary tag" followed by a *git merge*.

Fetching from a remote repository is done by, unsurprisingly, *git fetch*:

```
$ git fetch <remote-repository>
```

One of the following transports can be used to name the repository to download from:

SSH

remote.machine:/path/to/repo.git/ or

ssh://remote.machine/path/to/repo.git/

This transport can be used for both uploading and downloading, and requires you to have a log-in privilege over *ssh* to the remote machine. It finds out the set of objects the other side lacks by exchanging the head commits both ends have and transfers (close to) minimum set of objects. It is by far the most efficient way to exchange Git objects between repositories.

Local directory

/path/to/repo.git/

This transport is the same as SSH transport but uses *sh* to run both ends on the local machine instead of running other end on the remote machine via *ssh*.

Git Native

git://remote.machine/path/to/repo.git/

This transport was designed for anonymous downloading. Like SSH transport, it finds out the set of objects the downstream side lacks and transfers (close to) minimum set of objects.

HTTP(S)

http://remote.machine/path/to/repo.git/

Downloader from http and https URL first obtains the topmost commit object name from the remote site by looking at the specified refname under *repo.git/refs/* directory, and then tries to obtain the commit object by

downloading from *repo.git/objects/xx/xxx...* using the object name of that commit object. Then it reads the commit object to find out its parent commits and the associate tree object; it repeats this process until it gets all the necessary objects. Because of this behavior, they are sometimes also called *commit walkers*.

The *commit walkers* are sometimes also called *dumb transports*, because they do not require any Git aware smart server like Git Native transport does. Any stock HTTP server that does not even support directory index would suffice. But you must prepare your repository with *git update-server-info* to help dumb transport downloaders.

Once you fetch from the remote repository, you *merge* that with your current branch.

However -- it's such a common thing to *fetch* and then immediately *merge*, that it's called *git pull*, and you can simply do

```
$ git pull <remote-repository>
```

and optionally give a branch-name for the remote end as a second argument.

> **Note**
>
> You could do without using any branches at all, by keeping as many local repositories as you would like to have branches, and merging between them with *git pull*, just like you merge between branches. The advantage of this approach is that it lets you keep a set of files for each *branch* checked out and you may find it easier to switch back and forth if you juggle multiple lines of development simultaneously. Of course, you will pay the price of more disk usage to hold multiple working trees, but disk space is cheap these days.

It is likely that you will be pulling from the same remote repository from time to time. As a short hand, you can store the remote repository URL in the local repository's config file like this:

```
$ git config remote.linus.url https://git.kernel.org/pub/scm/git/git.git/
```

and use the "linus" keyword with *git pull* instead of the full URL.

Examples.

1. *git pull linus*
2. *git pull linus tag v0.99.1*

the above are equivalent to:

1. *git pull http://www.kernel.org/pub/scm/git/git.git/ HEAD*
2. *git pull http://www.kernel.org/pub/scm/git/git.git/ tag v0.99.1*

How does the merge work?

We said this tutorial shows what plumbing does to help you cope with the porcelain that isn't flushing, but we so far did not talk about how the merge really works. If you are following this tutorial the first time, I'd suggest to skip to "Publishing your work" section and come back here later.

OK, still with me? To give us an example to look at, let's go back to the earlier repository with "hello" and "example" file, and bring ourselves back to the pre-merge state:

```
$ git show-branch --more=2 master mybranch
! [master] Merge work in mybranch
 * [mybranch] Merge work in mybranch
--
-- [master] Merge work in mybranch
```

```
+* [master^2] Some work.
+* [master^] Some fun.
```

Remember, before running *git merge*, our *master* head was at "Some fun." commit, while our *mybranch* head was at "Some work." commit.

```
$ git switch -C mybranch master^2
$ git switch master
$ git reset --hard master^
```

After rewinding, the commit structure should look like this:

```
$ git show-branch
* [master] Some fun.
 ! [mybranch] Some work.
--
*  [master] Some fun.
 + [mybranch] Some work.
*+ [master^] Initial commit
```

Now we are ready to experiment with the merge by hand.

git merge command, when merging two branches, uses 3-way merge algorithm. First, it finds the common ancestor between them. The command it uses is *git merge-base*:

```
$ mb=$(git merge-base HEAD mybranch)
```

The command writes the commit object name of the common ancestor to the standard output, so we captured its output to a variable, because we will be using it in the next step. By the way, the common ancestor commit is the "Initial commit" commit in this case. You can tell it by:

```
$ git name-rev --name-only --tags $mb
my-first-tag
```

After finding out a common ancestor commit, the second step is this:

```
$ git read-tree -m -u $mb HEAD mybranch
```

This is the same *git read-tree* command we have already seen, but it takes three trees, unlike previous examples. This reads the contents of each tree into different *stage* in the index file (the first tree goes to stage 1, the second to stage 2, etc.). After reading three trees into three stages, the paths that are the same in all three stages are *collapsed* into stage 0. Also paths that are the same in two of three stages are collapsed into stage 0, taking the SHA-1 from either stage 2 or stage 3, whichever is different from stage 1 (i.e. only one side changed from the common ancestor).

After *collapsing* operation, paths that are different in three trees are left in non-zero stages. At this point, you can inspect the index file with this command:

```
$ git ls-files --stage
100644 7f8b141b65fdcee47321e399a2598a235a032422 0       example
100644 557db03de997c86a4a028e1ebd3a1ceb225be238 1       hello
100644 ba42a2a96e3027f3333e13ede4ccf4498c3ae942 2       hello
100644 cc44c73eb783565da5831b4d820c962954019b69 3       hello
```

In our example of only two files, we did not have unchanged files so only *example* resulted in collapsing. But in real-life large projects, when only a small number of files change in one commit, this *collapsing* tends to trivially merge most of the paths fairly quickly, leaving only a handful of real changes in non-zero stages.

To look at only non-zero stages, use *--unmerged* flag:

```
$ git ls-files --unmerged
100644 557db03de997c86a4a028e1ebd3a1ceb225be238 1       hello
```

```
100644 ba42a2a96e3027f3333e13ede4ccf4498c3ae942 2       hello
100644 cc44c73eb783565da5831b4d820c962954019b69 3       hello
```

The next step of merging is to merge these three versions of the file, using 3-way merge. This is done by giving *git merge-one-file* command as one of the arguments to *git merge-index* command:

```
$ git merge-index git-merge-one-file hello
Auto-merging hello
ERROR: Merge conflict in hello
fatal: merge program failed
```

git merge-one-file script is called with parameters to describe those three versions, and is responsible to leave the merge results in the working tree. It is a fairly straightforward shell script, and eventually calls *merge* program from RCS suite to perform a file-level 3-way merge. In this case, *merge* detects conflicts, and the merge result with conflict marks is left in the working tree.. This can be seen if you run *ls-files --stage* again at this point:

```
$ git ls-files --stage
100644 7f8b141b65fdcee47321e399a2598a235a032422 0       example
100644 557db03de997c86a4a028e1ebd3a1ceb225be238 1       hello
100644 ba42a2a96e3027f3333e13ede4ccf4498c3ae942 2       hello
100644 cc44c73eb783565da5831b4d820c962954019b69 3       hello
```

This is the state of the index file and the working file after *git merge* returns control back to you, leaving the conflicting merge for you to resolve. Notice that the path *hello* is still unmerged, and what you see with *git diff* at this point is differences since stage 2 (i.e. your version).

Publishing your work

So, we can use somebody else's work from a remote repository, but how can **you** prepare a repository to let other people pull from it?

You do your real work in your working tree that has your primary repository hanging under it as its *.git* subdirectory. You **could** make that repository accessible remotely and ask people to pull from it, but in practice that is not the way things are usually done. A recommended way is to have a public repository, make it reachable by other people, and when the changes you made in your primary working tree are in good shape, update the public repository from it. This is often called *pushing*.

> **Note**
>
> This public repository could further be mirrored, and that is how Git repositories at *kernel.org* are managed.

Publishing the changes from your local (private) repository to your remote (public) repository requires a write privilege on the remote machine. You need to have an SSH account there to run a single command, *git-receive-pack*.

First, you need to create an empty repository on the remote machine that will house your public repository. This empty repository will be populated and be kept up to date by pushing into it later. Obviously, this repository creation needs to be done only once.

> **Note**
>
> *git push* uses a pair of commands, *git send-pack* on your local machine, and *git-receive-pack* on the remote machine. The communication between the two over the network internally uses an SSH connection.

Your private repository's Git directory is usually *.git*, but your public repository is often named after the project name, i.e. *<project>.git*. Let's create such a public repository for project *my-git*. After logging into the remote machine, create an empty directory:

```
$ mkdir my-git.git
```

Then, make that directory into a Git repository by running *git init*, but this time, since its name is not the usual *.git*, we do things slightly differently:

```
$ GIT_DIR=my-git.git git init
```

Make sure this directory is available for others you want your changes to be pulled via the transport of your choice. Also you need to make sure that you have the *git-receive-pack* program on the *$PATH*.

> **Note**
>
> Many installations of sshd do not invoke your shell as the login shell when you directly run programs; what this means is that if your login shell is *bash*, only *.bashrc* is read and not *.bash_profile*. As a workaround, make sure *.bashrc* sets up *$PATH* so that you can run *git-receive-pack* program.

> **Note**
>
> If you plan to publish this repository to be accessed over http, you should do *mv my-git.git/hooks/post-update.sample my-git.git/hooks/post-update* at this point. This makes sure that every time you push into this repository, *git update-server-info* is run.

Your "public repository" is now ready to accept your changes. Come back to the machine you have your private repository. From there, run this command:

```
$ git push <public-host>:/path/to/my-git.git master
```

This synchronizes your public repository to match the named branch head (i.e. *master* in this case) and objects reachable from them in your current repository.

As a real example, this is how I update my public Git repository. Kernel.org mirror network takes care of the propagation to other publicly visible machines:

```
$ git push master.kernel.org:/pub/scm/git/git.git/
```

Packing your repository

Earlier, we saw that one file under *.git/objects/??/* directory is stored for each Git object you create. This representation is efficient to create atomically and safely, but not so convenient to transport over the network. Since Git objects are immutable once they are created, there is a way to optimize the storage by "packing them together". The command

```
$ git repack
```

will do it for you. If you followed the tutorial examples, you would have accumulated about 17 objects in *.git/objects/??/* directories by now. *git repack* tells you how many objects it packed, and stores the packed file in the *.git/objects/pack* directory.

> **Note**
>
> You will see two files, *pack-*.pack* and *pack-*.idx*, in *.git/objects/pack* directory. They are closely related to each other, and if you ever copy them by hand to a different repository for whatever reason, you should make sure you copy them together. The former holds all the data from the objects in the pack, and the latter holds the index for random access.

If you are paranoid, running *git verify-pack* command would detect if you have a corrupt pack, but do not worry too much. Our programs are always perfect ;-).

Once you have packed objects, you do not need to leave the unpacked objects that are contained in the pack file anymore.

```
$ git prune-packed
```

would remove them for you.

You can try running *find .git/objects -type f* before and after you run *git prune-packed* if you are curious. Also *git count-objects* would tell you how many unpacked objects are in your repository and how much space they are consuming.

> **Note**
>
> *git pull* is slightly cumbersome for HTTP transport, as a packed repository may contain relatively few objects in a relatively large pack. If you expect many HTTP pulls from your public repository you might want to repack & prune often, or never.

If you run *git repack* again at this point, it will say "Nothing new to pack.". Once you continue your development and accumulate the changes, running *git repack* again will create a new pack, that contains objects created since you packed your repository the last time. We recommend that you pack your project soon after the initial import (unless you are starting your project from scratch), and then run *git repack* every once in a while, depending on how active your project is.

When a repository is synchronized via *git push* and *git pull* objects packed in the source repository are usually stored unpacked in the destination. While this allows you to use different packing strategies on both ends, it also means you may need to repack both repositories every once in a while.

Working with Others

Although Git is a truly distributed system, it is often convenient to organize your project with an informal hierarchy of developers. Linux kernel development is run this way. There is a nice illustration (page 17, "Merges to Mainline") in Randy Dunlap's presentation [https://web.archive.org/web/20120915203609/http://www.xenotime.net/linux/mentor/linux-mentoring-2006.pdf].

It should be stressed that this hierarchy is purely **informal**. There is nothing fundamental in Git that enforces the "chain of patch flow" this hierarchy implies. You do not have to pull from only one remote repository.

A recommended workflow for a "project lead" goes like this:

1. Prepare your primary repository on your local machine. Your work is done there.

2. Prepare a public repository accessible to others.

   If other people are pulling from your repository over dumb transport protocols (HTTP), you need to keep this repository *dumb transport friendly*. After *git init*, *$GIT_DIR/hooks/post-update.sample* copied from the standard templates would contain a call to *git update-server-info* but you need to manually enable the hook with *mv post-update.sample post-update*. This makes sure *git update-server-info* keeps the necessary files up to date.

3. Push into the public repository from your primary repository.

4. *git repack* the public repository. This establishes a big pack that contains the initial set of objects as the baseline, and possibly *git prune* if the transport used for pulling from your repository supports packed repositories.

5. Keep working in your primary repository. Your changes include modifications of your own, patches you receive via e-mails, and merges resulting from pulling the "public" repositories of your "subsystem maintainers".

   You can repack this private repository whenever you feel like.

6. Push your changes to the public repository, and announce it to the public.

7. Every once in a while, *git repack* the public repository. Go back to step 5. and continue working.

A recommended work cycle for a "subsystem maintainer" who works on that project and has an own "public repository" goes like this:

1. Prepare your work repository, by running *git clone* on the public repository of the "project lead". The URL used for the initial cloning is stored in the remote.origin.url configuration variable.

2. Prepare a public repository accessible to others, just like the "project lead" person does.

3. Copy over the packed files from "project lead" public repository to your public repository, unless the "project lead" repository lives on the same machine as yours. In the latter case, you can use *objects/info/alternates* file to point at the repository you are borrowing from.

4. Push into the public repository from your primary repository. Run *git repack*, and possibly *git prune* if the transport used for pulling from your repository supports packed repositories.

5. Keep working in your primary repository. Your changes include modifications of your own, patches you receive via e-mails, and merges resulting from pulling the "public" repositories of your "project lead" and possibly your "sub-subsystem maintainers".

   You can repack this private repository whenever you feel like.

6. Push your changes to your public repository, and ask your "project lead" and possibly your "sub-subsystem maintainers" to pull from it.

7. Every once in a while, *git repack* the public repository. Go back to step 5. and continue working.

A recommended work cycle for an "individual developer" who does not have a "public" repository is somewhat different. It goes like this:

1. Prepare your work repository, by *git clone* the public repository of the "project lead" (or a "subsystem maintainer", if you work on a subsystem). The URL used for the initial cloning is stored in the remote.origin.url configuration variable.

2. Do your work in your repository on *master* branch.

3. Run *git fetch origin* from the public repository of your upstream every once in a while. This does only the first half of *git pull* but does not merge. The head of the public repository is stored in *.git/refs/remotes/origin/master*.

4. Use *git cherry origin* to see which ones of your patches were accepted, and/or use *git rebase origin* to port your unmerged changes forward to the updated upstream.

5. Use *git format-patch origin* to prepare patches for e-mail submission to your upstream and send it out. Go back to step 2. and continue.

Working with Others, Shared Repository Style

If you are coming from a CVS background, the style of cooperation suggested in the previous section may be new to you. You do not have to worry. Git supports the "shared public repository" style of cooperation you are probably more familiar with as well.

See Section G.2.4, “gitcvs-migration(7)” for the details.

Bundling your work together

It is likely that you will be working on more than one thing at a time. It is easy to manage those more-or-less independent tasks using branches with Git.

We have already seen how branches work previously, with "fun and work" example using two branches. The idea is the same if there are more than two branches. Let's say you started out from "master" head, and have some new code in the "master" branch, and two independent fixes in the "commit-fix" and "diff-fix" branches:

```
$ git show-branch
! [commit-fix] Fix commit message normalization.
 ! [diff-fix] Fix rename detection.
  * [master] Release candidate #1
---
 +  [diff-fix] Fix rename detection.
 +  [diff-fix~1] Better common substring algorithm.
+   [commit-fix] Fix commit message normalization.
  * [master] Release candidate #1
++* [diff-fix~2] Pretty-print messages.
```

Both fixes are tested well, and at this point, you want to merge in both of them. You could merge in *diff-fix* first and then *commit-fix* next, like this:

```
$ git merge -m "Merge fix in diff-fix" diff-fix
$ git merge -m "Merge fix in commit-fix" commit-fix
```

Which would result in:

```
$ git show-branch
! [commit-fix] Fix commit message normalization.
 ! [diff-fix] Fix rename detection.
  * [master] Merge fix in commit-fix
---
  - [master] Merge fix in commit-fix
+ * [commit-fix] Fix commit message normalization.
  - [master~1] Merge fix in diff-fix
 +* [diff-fix] Fix rename detection.
 +* [diff-fix~1] Better common substring algorithm.
  * [master~2] Release candidate #1
++* [master~3] Pretty-print messages.
```

However, there is no particular reason to merge in one branch first and the other next, when what you have are a set of truly independent changes (if the order mattered, then they are not independent by definition). You could instead merge those two branches into the current branch at once. First let's undo what we just did and start over. We would want to get the master branch before these two merges by resetting it to *master~2*:

```
$ git reset --hard master~2
```

You can make sure *git show-branch* matches the state before those two *git merge* you just did. Then, instead of running two *git merge* commands in a row, you would merge these two branch heads (this is known as *making an Octopus*):

```
$ git merge commit-fix diff-fix
$ git show-branch
! [commit-fix] Fix commit message normalization.
 ! [diff-fix] Fix rename detection.
  * [master] Octopus merge of branches 'diff-fix' and 'commit-fix'
---
  - [master] Octopus merge of branches 'diff-fix' and 'commit-fix'
+ * [commit-fix] Fix commit message normalization.
 +* [diff-fix] Fix rename detection.
 +* [diff-fix~1] Better common substring algorithm.
  * [master~1] Release candidate #1
++* [master~2] Pretty-print messages.
```

Note that you should not do Octopus just because you can. An octopus is a valid thing to do and often makes it easier to view the commit history if you are merging more than two independent changes at the same time. However, if you have merge conflicts with any of the branches you are merging in and need to hand resolve, that is an indication that the development happened in those branches were not independent after all, and you should

merge two at a time, documenting how you resolved the conflicts, and the reason why you preferred changes made in one side over the other. Otherwise it would make the project history harder to follow, not easier.

SEE ALSO

Section G.2.1, “gittutorial(7)”, Section G.2.2, “gittutorial-2(7)”, Section G.2.4, “gitcvs-migration(7)”, Section G.3.63, “git-help(1)”, Section G.2.5, “giteveryday(7)”, *The Git User's Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html]

GIT

Part of the Section G.3.1, “git(1)” suite

G.2.4. gitcvs-migration(7)

2024-07-02

Revision History

2024-07-02

NAME

gitcvs-migration - Git for CVS users

SYNOPSIS

*git cvsimport **

DESCRIPTION

Git differs from CVS in that every working tree contains a repository with a full copy of the project history, and no repository is inherently more important than any other. However, you can emulate the CVS model by designating a single shared repository which people can synchronize with; this document explains how to do that.

Some basic familiarity with Git is required. Having gone through Section G.2.1, “gittutorial(7)” and Section G.4.19, “gitglossary(7)” should be sufficient.

Developing against a shared repository

Suppose a shared repository is set up in /pub/repo.git on the host foo.com. Then as an individual committer you can clone the shared repository over ssh with:

```
$ git clone foo.com:/pub/repo.git/ my-project
$ cd my-project
```

and hack away. The equivalent of *cvs update* is

```
$ git pull origin
```

which merges in any work that others might have done since the clone operation. If there are uncommitted changes in your working tree, commit them first before running git pull.

> **Note**
>
> The *pull* command knows where to get updates from because of certain configuration variables that were set by the first *git clone* command; see *git config -l* and the Section G.3.29, “git-config(1)” man page for details.

You can update the shared repository with your changes by first committing your changes, and then using the *git push* command:

```
$ git push origin master
```

to "push" those commits to the shared repository. If someone else has updated the repository more recently, *git push*, like *cvs commit*, will complain, in which case you must pull any changes before attempting the push again.

In the *git push* command above we specify the name of the remote branch to update (*master*). If we leave that out, *git push* tries to update any branches in the remote repository that have the same name as a branch in the local repository. So the last *push* can be done with either of:

```
$ git push origin
$ git push foo.com:/pub/project.git/
```

as long as the shared repository does not have any branches other than *master*.

Setting Up a Shared Repository

We assume you have already created a Git repository for your project, possibly created from scratch or from a tarball (see Section G.2.1, “gittutorial(7)”), or imported from an already existing CVS repository (see the next section).

Assume your existing repo is at /home/alice/myproject. Create a new "bare" repository (a repository without a working tree) and fetch your project into it:

```
$ mkdir /pub/my-repo.git
$ cd /pub/my-repo.git
$ git --bare init --shared
$ git --bare fetch /home/alice/myproject master:master
```

Next, give every team member read/write access to this repository. One easy way to do this is to give all the team members ssh access to the machine where the repository is hosted. If you don't want to give them a full shell on the machine, there is a restricted shell which only allows users to do Git pushes and pulls; see Section G.3.129, “git-shell(1)”.

Put all the committers in the same group, and make the repository writable by that group:

```
$ chgrp -R $group /pub/my-repo.git
```

Make sure committers have a umask of at most 027, so that the directories they create are writable and searchable by other group members.

Importing a CVS archive

> **Note**
>
> These instructions use the *git-cvsimport* script which ships with git, but other importers may provide better results. See the note in Section G.3.36, “git-cvsimport(1)” for other options.

First, install version 2.1 or higher of cvsps from https://github.com/andreyvit/cvsps and make sure it is in your path. Then cd to a checked out CVS working directory of the project you are interested in and run Section G.3.36, “git-cvsimport(1)”:

```
$ git cvsimport -C <destination> <module>
```

This puts a Git archive of the named CVS module in the directory <destination>, which will be created if necessary.

The import checks out from CVS every revision of every file. Reportedly cvsimport can average some twenty revisions per second, so for a medium-sized project this should not take more than a couple of minutes. Larger projects or remote repositories may take longer.

The main trunk is stored in the Git branch named *origin*, and additional CVS branches are stored in Git branches with the same names. The most recent version of the main trunk is also left checked out on the *master* branch, so you can start adding your own changes right away.

The import is incremental, so if you call it again next month it will fetch any CVS updates that have been made in the meantime. For this to work, you must not modify the imported branches; instead, create new branches for your own changes, and merge in the imported branches as necessary.

If you want a shared repository, you will need to make a bare clone of the imported directory, as described above. Then treat the imported directory as another development clone for purposes of merging incremental imports.

Advanced Shared Repository Management

Git allows you to specify scripts called "hooks" to be run at certain points. You can use these, for example, to send all commits to the shared repository to a mailing list. See Section G.4.7, “githooks(5)”.

You can enforce finer grained permissions using update hooks. See *Controlling access to branches using update hooks* [https://www.kernel.org/pub/software/scm/git/docs/howto/update-hook-example.html].

Providing CVS Access to a Git Repository

It is also possible to provide true CVS access to a Git repository, so that developers can still use CVS; see Section G.3.37, “git-cvsserver(1)” for details.

Alternative Development Models

CVS users are accustomed to giving a group of developers commit access to a common repository. As we've seen, this is also possible with Git. However, the distributed nature of Git allows other development models, and you may want to first consider whether one of them might be a better fit for your project.

For example, you can choose a single person to maintain the project's primary public repository. Other developers then clone this repository and each work in their own clone. When they have a series of changes that they're happy with, they ask the maintainer to pull from the branch containing the changes. The maintainer reviews their changes and pulls them into the primary repository, which other developers pull from as necessary to stay coordinated. The Linux kernel and other projects use variants of this model.

With a small group, developers may just pull changes from each other's repositories without the need for a central maintainer.

SEE ALSO

Section G.2.1, “gittutorial(7)”, Section G.2.2, “gittutorial-2(7)”, Section G.2.3, “gitcore-tutorial(7)”, Section G.4.19, “gitglossary(7)”, Section G.2.5, “giteveryday(7)”, *The Git User's Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html]

GIT

Part of the Section G.3.1, “git(1)” suite

G.2.5. giteveryday(7)

2024-07-02

Revision History

2024-07-02

NAME

giteveryday - A useful minimum set of commands for Everyday Git

SYNOPSIS

Everyday Git With 20 Commands Or So

DESCRIPTION

Git users can broadly be grouped into four categories for the purposes of describing here a small set of useful commands for everyday Git.

- Individual Developer (Standalone) commands are essential for anybody who makes a commit, even for somebody who works alone.
- If you work with other people, you will need commands listed in the Individual Developer (Participant) section as well.
- People who play the Integrator role need to learn some more commands in addition to the above.
- Repository Administration commands are for system administrators who are responsible for the care and feeding of Git repositories.

Individual Developer (Standalone)

A standalone individual developer does not exchange patches with other people, and works alone in a single repository, using the following commands.

- Section G.3.71, “git-init(1)” to create a new repository.
- Section G.3.74, “git-log(1)” to see what happened.
- Section G.3.140, “git-switch(1)” and Section G.3.10, “git-branch(1)” to switch branches.
- Section G.3.2, “git-add(1)” to manage the index file.
- Section G.3.44, “git-diff(1)” and Section G.3.138, “git-status(1)” to see what you are in the middle of doing.
- Section G.3.28, “git-commit(1)” to advance the current branch.
- Section G.3.119, “git-restore(1)” to undo changes.
- Section G.3.86, “git-merge(1)” to merge between local branches.
- Section G.3.107, “git-rebase(1)” to maintain topic branches.
- Section G.3.144, “git-tag(1)” to mark a known point.

1. Examples

Use a tarball as a starting point for a new repository.

```
$ tar zxf frotz.tar.gz
$ cd frotz
$ git init
$ git add . ❶
$ git commit -m "import of frotz source tree."
$ git tag v2.43 ❷
```

❶ add everything under the current directory.
❷ make a lightweight, unannotated tag.

Create a topic branch and develop.

```
$ git switch -c alsa-audio ❶
$ edit/compile/test
$ git restore curses/ux_audio_oss.c ❷
$ git add curses/ux_audio_alsa.c ❸
$ edit/compile/test
$ git diff HEAD ❹
$ git commit -a -s ❺
$ edit/compile/test
$ git diff HEAD^ ❻
```

```
$ git commit -a --amend ❼
$ git switch master ❽
$ git merge alsa-audio ❾
$ git log --since='3 days ago' ❿
$ git log v2.43.. curses/ ⓫
```

❶ create a new topic branch.
❷ revert your botched changes in *curses/ux_audio_oss.c*.
❸ you need to tell Git if you added a new file; removal and modification will be caught if you do *git commit -a* later.
❹ to see what changes you are committing.
❺ commit everything, as you have tested, with your sign-off.
❻ look at all your changes including the previous commit.
❼ amend the previous commit, adding all your new changes, using your original message.
❽ switch to the master branch.
❾ merge a topic branch into your master branch.
❿ review commit logs; other forms to limit output can be combined and include *-10* (to show up to 10 commits), *--until=2005-12-10*, etc.
⓫ view only the changes that touch what's in *curses/* directory, since *v2.43* tag.

Individual Developer (Participant)

A developer working as a participant in a group project needs to learn how to communicate with others, and uses these commands in addition to the ones needed by a standalone developer.

- Section G.3.24, “git-clone(1)” from the upstream to prime your local repository.
- Section G.3.102, “git-pull(1)” and Section G.3.49, “git-fetch(1)” from "origin" to keep up-to-date with the upstream.
- Section G.3.103, “git-push(1)” to shared repository, if you adopt CVS style shared repository workflow.
- Section G.3.54, “git-format-patch(1)” to prepare e-mail submission, if you adopt Linux kernel-style public forum workflow.
- Section G.3.124, “git-send-email(1)” to send your e-mail submission without corruption by your MUA.
- Section G.3.116, “git-request-pull(1)” to create a summary of changes for your upstream to pull.

1. Examples

Clone the upstream and work on it. Feed changes to upstream.

```
$ git clone git://git.kernel.org/pub/scm/.../torvalds/linux-2.6 my2.6
$ cd my2.6
$ git switch -c mine master ❶
$ edit/compile/test; git commit -a -s ❷
$ git format-patch master ❸
$ git send-email --to="person <email@example.com>" 00*.patch ❹
$ git switch master ❺
$ git pull ❻
$ git log -p ORIG_HEAD.. arch/i386 include/asm-i386 ❼
$ git ls-remote --heads http://git.kernel.org/.../jgarzik/libata-
dev.git ❽
$ git pull git://git.kernel.org/pub/.../jgarzik/libata-dev.git ALL ❾
$ git reset --hard ORIG_HEAD ❿
$ git gc ⓫
```

❶ checkout a new branch *mine* from master.

❷ repeat as needed.
❸ extract patches from your branch, relative to master,
❹ and email them.
❺ return to *master*, ready to see what's new
❻ *git pull* fetches from *origin* by default and merges into the current branch.
❼ immediately after pulling, look at the changes done upstream since last time we checked, only in the area we are interested in.
❽ check the branch names in an external repository (if not known).
❾ fetch from a specific branch *ALL* from a specific repository and merge it.
❿ revert the pull.
⓫ garbage collect leftover objects from reverted pull.

Push into another repository.

```
satellite$ git clone mothership:frotz frotz ❶
satellite$ cd frotz
satellite$ git config --get-regexp '^(remote|branch)\.' ❷
remote.origin.url mothership:frotz
remote.origin.fetch refs/heads/*:refs/remotes/origin/*
branch.master.remote origin
branch.master.merge refs/heads/master
satellite$ git config remote.origin.push \
           +refs/heads/*:refs/remotes/satellite/* ❸
satellite$ edit/compile/test/commit
satellite$ git push origin ❹

mothership$ cd frotz
mothership$ git switch master
mothership$ git merge satellite/master ❺
```

❶ mothership machine has a frotz repository under your home directory; clone from it to start a repository on the satellite machine.
❷ clone sets these configuration variables by default. It arranges *git pull* to fetch and store the branches of mothership machine to local *remotes/origin/** remote-tracking branches.
❸ arrange *git push* to push all local branches to their corresponding branch of the mothership machine.
❹ push will stash all our work away on *remotes/satellite/** remote-tracking branches on the mothership machine. You could use this as a back-up method. Likewise, you can pretend that mothership "fetched" from you (useful when access is one sided).
❺ on mothership machine, merge the work done on the satellite machine into the master branch.

Branch off of a specific tag.

```
$ git switch -c private2.6.14 v2.6.14 ❶
$ edit/compile/test; git commit -a
$ git checkout master
$ git cherry-pick v2.6.14..private2.6.14 ❷
```

❶ create a private branch based on a well known (but somewhat behind) tag.
❷ forward port all changes in *private2.6.14* branch to *master* branch without a formal "merging". Or long-hand *git format-patch -k -m --stdout v2.6.14..private2.6.14 | git am -3 -k*

An alternate participant submission mechanism is using the *git request-pull* or pull-request mechanisms (e.g. as used on GitHub (www.github.com) to notify your upstream of your contribution.

Integrator

A fairly central person acting as the integrator in a group project receives changes made by others, reviews and integrates them and publishes the result for others to use, using these commands in addition to the ones needed by participants.

This section can also be used by those who respond to *git request-pull* or pull-request on GitHub (www.github.com) to integrate the work of others into their history. A sub-area lieutenant for a repository will act both as a participant and as an integrator.

- Section G.3.3, “git-am(1)” to apply patches e-mailed in from your contributors.
- Section G.3.102, “git-pull(1)” to merge from your trusted lieutenants.
- Section G.3.54, “git-format-patch(1)” to prepare and send suggested alternative to contributors.
- Section G.3.122, “git-revert(1)” to undo botched commits.
- Section G.3.103, “git-push(1)” to publish the bleeding edge.

1. Examples

A typical integrator's Git day.

```
$ git status ❶
$ git branch --no-merged master ❷
$ mailx ❸
& s 2 3 4 5 ./+to-apply
& s 7 8 ./+hold-linus
& q
$ git switch -c topic/one master
$ git am -3 -i -s ./+to-apply ❹
$ compile/test
$ git switch -c hold/linus && git am -3 -i -s ./+hold-linus ❺
$ git switch topic/one && git rebase master ❻
$ git switch -C seen next ❼
$ git merge topic/one topic/two && git merge hold/linus ❽
$ git switch maint
$ git cherry-pick master~4 ❾
$ compile/test
$ git tag -s -m "GIT 0.99.9x" v0.99.9x ❿
$ git fetch ko && for branch in master maint next seen ⓫
    do
        git show-branch ko/$branch $branch ⓬
    done
$ git push --follow-tags ko ⓭
```

❶ see what you were in the middle of doing, if anything.
❷ see which branches haven't been merged into *master* yet. Likewise for any other integration branches e.g. *maint*, *next* and *seen*.
❸ read mails, save ones that are applicable, and save others that are not quite ready (other mail readers are available).
❹ apply them, interactively, with your sign-offs.
❺ create topic branch as needed and apply, again with sign-offs.
❻ rebase internal topic branch that has not been merged to the master or exposed as a part of a stable branch.
❼ restart *seen* every time from the next.
❽ and bundle topic branches still cooking.
❾ backport a critical fix.
❿ create a signed tag.
⓫ make sure master was not accidentally rewound beyond that already pushed out.
⓬ In the output from *git show-branch*, *master* should have everything *ko/master* has, and *next* should have everything *ko/next* has, etc.
⓭ push out the bleeding edge, together with new tags that point into the pushed history.

In this example, the *ko* shorthand points at the Git maintainer's repository at kernel.org, and looks like this:

```
(in .git/config)
[remote "ko"]
        url = kernel.org:/pub/scm/git/git.git
        fetch = refs/heads/*:refs/remotes/ko/*
        push = refs/heads/master
        push = refs/heads/next
        push = +refs/heads/seen
        push = refs/heads/maint
```

Repository Administration

A repository administrator uses the following tools to set up and maintain access to the repository by developers.

- Section G.3.38, “git-daemon(1)” to allow anonymous download from repository.
- Section G.3.129, “git-shell(1)” can be used as a *restricted login shell* for shared central repository users.
- Section G.3.65, “git-http-backend(1)” provides a server side implementation of Git-over-HTTP ("Smart http") allowing both fetch and push services.
- Section G.4.16, “gitweb(1)” provides a web front-end to Git repositories, which can be set-up using the Section G.3.72, “git-instaweb(1)” script.

update hook howto [https://www.kernel.org/pub/software/scm/git/docs/howto/update-hook-example.html] has a good example of managing a shared central repository.

In addition there are a number of other widely deployed hosting, browsing and reviewing solutions such as:

- gitolite, gerrit code review, cgit and others.

1. Examples

We assume the following in /etc/services

```
$ grep 9418 /etc/services
git             9418/tcp                # Git Version Control System
```

Run git-daemon to serve /pub/scm from inetd.

```
$ grep git /etc/inetd.conf
git     stream  tcp     nowait  nobody \
  /usr/bin/git-daemon git-daemon --inetd --export-all /pub/scm
```

The actual configuration line should be on one line.

Run git-daemon to serve /pub/scm from xinetd.

```
$ cat /etc/xinetd.d/git-daemon
# default: off
# description: The Git server offers access to Git repositories
service git
{
        disable = no
        type            = UNLISTED
        port            = 9418
        socket_type     = stream
        wait            = no
        user            = nobody
        server          = /usr/bin/git-daemon
        server_args     = --inetd --export-all --base-path=/pub/scm
```

```
        log_on_failure  += USERID
}
```

Check your xinetd(8) documentation and setup, this is from a Fedora system. Others might be different.

Give push/pull only access to developers using git-over-ssh.

e.g. those using: *$ git push/pull ssh://host.xz/pub/scm/project*

```
$ grep git /etc/passwd ❶
alice:x:1000:1000::/home/alice:/usr/bin/git-shell
bob:x:1001:1001::/home/bob:/usr/bin/git-shell
cindy:x:1002:1002::/home/cindy:/usr/bin/git-shell
david:x:1003:1003::/home/david:/usr/bin/git-shell
$ grep git /etc/shells ❷
/usr/bin/git-shell
```

❶ log-in shell is set to /usr/bin/git-shell, which does not allow anything but *git push* and *git pull*. The users require ssh access to the machine.
❷ in many distributions /etc/shells needs to list what is used as the login shell.

CVS-style shared repository.

```
$ grep git /etc/group ❶
git:x:9418:alice,bob,cindy,david
$ cd /home/devo.git
$ ls -l ❷
  lrwxrwxrwx   1 david git    17 Dec  4 22:40 HEAD -> refs/heads/master
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 branches
  -rw-rw-r--   1 david git    84 Dec  4 22:40 config
  -rw-rw-r--   1 david git    58 Dec  4 22:40 description
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 hooks
  -rw-rw-r--   1 david git 37504 Dec  4 22:40 index
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 info
  drwxrwsr-x   4 david git  4096 Dec  4 22:40 objects
  drwxrwsr-x   4 david git  4096 Nov  7 14:58 refs
  drwxrwsr-x   2 david git  4096 Dec  4 22:40 remotes
$ ls -l hooks/update ❸
  -r-xr-xr-x   1 david git  3536 Dec  4 22:40 update
$ cat info/allowed-users ❹
refs/heads/master       alice\|cindy
refs/heads/doc-update   bob
refs/tags/v[0-9]*       david
```

❶ place the developers into the same git group.
❷ and make the shared repository writable by the group.
❸ use update-hook example by Carl from Documentation/howto/ for branch policy control.
❹ alice and cindy can push into master, only bob can push into doc-update. david is the release manager and is the only person who can create and push version tags.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3. Git Command Reference

G.3.1. git(1)

2024-07-02
Revision History

2024-07-02

NAME

git - the stupid content tracker

SYNOPSIS

git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
[--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
[-p|--paginate|-P|--no-pager] [--no-replace-objects] [--bare]
[--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
[--config-env=<name>=<envvar>] <command> [<args>]

DESCRIPTION

Git is a fast, scalable, distributed revision control system with an unusually rich command set that provides both high-level operations and full access to internals.

See Section G.2.1, “gittutorial(7)” to get started, then see Section G.2.5, “giteveryday(7)” for a useful minimum set of commands. The *Git User's Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html] has a more in-depth introduction.

After you mastered the basic concepts, you can come back to this page to learn what commands Git offers. You can learn more about individual Git commands with "git help command". Section G.4.1, “gitcli(7)” manual page gives you an overview of the command-line command syntax.

A formatted and hyperlinked copy of the latest Git documentation can be viewed at https://git.github.io/html-docs/git.html or https://git-scm.com/docs.

OPTIONS

-v , --version

Prints the Git suite version that the *git* program came from.

This option is internally converted to *git version* ... and accepts the same options as the Section G.3.156, “git-version(1)” command. If *--help* is also given, it takes precedence over *--version*.

-h , --help

Prints the synopsis and a list of the most commonly used commands. If the option *--all* or *-a* is given then all available commands are printed. If a Git command is named this option will bring up the manual page for that command.

Other options are available to control how the manual page is displayed. See Section G.3.63, “git-help(1)” for more information, because *git --help* ... is converted internally into *git help* ....

-C <path>

Run as if git was started in *<path>* instead of the current working directory. When multiple *-C* options are given, each subsequent non-absolute *-C <path>* is interpreted relative to the preceding *-C <path>*. If *<path>* is present but empty, e.g. *-C ""*, then the current working directory is left unchanged.

This option affects options that expect path name like *--git-dir* and *--work-tree* in that their interpretations of the path names would be made relative to the working directory caused by the *-C* option. For example the following invocations are equivalent:

```
git --git-dir=a.git --work-tree=b -C c status
git --git-dir=c/a.git --work-tree=c/b status
```

-c <name>=<value>

Pass a configuration parameter to the command. The value given will override values from configuration files. The <name> is expected in the same format as listed by *git config* (subkeys separated by dots).

Note that omitting the = in *git -c foo.bar* ... is allowed and sets *foo.bar* to the boolean true value (just like *[foo]bar* would in a config file). Including the equals but with an empty value (like *git -c foo.bar=* ...) sets *foo.bar* to the empty string which *git config --type=bool* will convert to *false*.

--config-env=<name>=<envvar>

Like *-c <name>=<value>*, give configuration variable *<name>* a value, where <envvar> is the name of an environment variable from which to retrieve the value. Unlike *-c* there is no shortcut for directly setting the value to an empty string, instead the environment variable itself must be set to the empty string. It is an error if the *<envvar>* does not exist in the environment. *<envvar>* may not contain an equals sign to avoid ambiguity with *<name>* containing one.

This is useful for cases where you want to pass transitory configuration options to git, but are doing so on operating systems where other processes might be able to read your command line (e.g. */proc/self/cmdline*), but not your environment (e.g. */proc/self/environ*). That behavior is the default on Linux, but may not be on your system.

Note that this might add security for variables such as *http.extraHeader* where the sensitive information is part of the value, but not e.g. *url.<base>.insteadOf* where the sensitive information can be part of the key.

--exec-path[=<path>]

Path to wherever your core Git programs are installed. This can also be controlled by setting the GIT_EXEC_PATH environment variable. If no path is given, *git* will print the current setting and then exit.

--html-path

Print the path, without trailing slash, where Git's HTML documentation is installed and exit.

--man-path

Print the manpath (see *man(1)*) for the man pages for this version of Git and exit.

--info-path

Print the path where the Info files documenting this version of Git are installed and exit.

-p , --paginate

Pipe all output into *less* (or if set, $PAGER) if standard output is a terminal. This overrides the *pager.<cmd>* configuration options (see the "Configuration Mechanism" section below).

-P , --no-pager

Do not pipe Git output into a pager.

--git-dir=<path>

Set the path to the repository (".git" directory). This can also be controlled by setting the *GIT_DIR* environment variable. It can be an absolute path or relative path to current working directory.

Specifying the location of the ".git" directory using this option (or *GIT_DIR* environment variable) turns off the repository discovery that tries to find a directory with ".git" subdirectory (which is how the repository and the top-level of the working tree are discovered), and tells Git that you are at the top level of the working tree. If you are not at the top-level directory of the working tree, you should tell Git where the top-level of the working tree is, with the *--work-tree=<path>* option (or *GIT_WORK_TREE* environment variable)

If you just want to run git as if it was started in *<path>* then use *git -C <path>*.

--work-tree=<path>

Set the path to the working tree. It can be an absolute path or a path relative to the current working directory. This can also be controlled by setting the GIT_WORK_TREE environment variable and the core.worktree configuration variable (see core.worktree in Section G.3.29, “git-config(1)” for a more detailed discussion).

--namespace=<path>

Set the Git namespace. See Section G.4.11, “gitnamespaces(7)” for more details. Equivalent to setting the *GIT_NAMESPACE* environment variable.

--bare

Treat the repository as a bare repository. If GIT_DIR environment is not set, it is set to the current working directory.

--no-replace-objects

Do not use replacement refs to replace Git objects. This is equivalent to exporting the *GIT_NO_REPLACE_OBJECTS* environment variable with any value. See Section G.3.114, “git-replace(1)” for more information.

--no-lazy-fetch

Do not fetch missing objects from the promisor remote on demand. Useful together with *git cat-file -e <object>* to see if the object is locally available. This is equivalent to setting the *GIT_NO_LAZY_FETCH* environment variable to *1*.

--literal-pathspecs

Treat pathspecs literally (i.e. no globbing, no pathspec magic). This is equivalent to setting the *GIT_LITERAL_PATHSPECS* environment variable to *1*.

--glob-pathspecs

Add "glob" magic to all pathspec. This is equivalent to setting the *GIT_GLOB_PATHSPECS* environment variable to *1*. Disabling globbing on individual pathspecs can be done using pathspec magic ":(literal)"

--noglob-pathspecs

Add "literal" magic to all pathspec. This is equivalent to setting the *GIT_NOGLOB_PATHSPECS* environment variable to *1*. Enabling globbing on individual pathspecs can be done using pathspec magic ":(glob)"

--icase-pathspecs

Add "icase" magic to all pathspec. This is equivalent to setting the *GIT_ICASE_PATHSPECS* environment variable to *1*.

--no-optional-locks

Do not perform optional operations that require locks. This is equivalent to setting the *GIT_OPTIONAL_LOCKS* to *0*.

--list-cmds=<group>[,<group>…]

List commands by group. This is an internal/experimental option and may change or be removed in the future. Supported groups are: builtins, parseopt (builtin commands that use parse-options), main (all commands in libexec directory), others (all other commands in *$PATH* that have git- prefix), list-<category> (see categories in command-list.txt), nohelpers (exclude helper commands), alias and config (retrieve command list from config variable completion.commands)

--attr-source=<tree-ish>

Read gitattributes from <tree-ish> instead of the worktree. See Section G.4.2, “gitattributes(5)”. This is equivalent to setting the *GIT_ATTR_SOURCE* environment variable.

GIT COMMANDS

We divide Git into high level ("porcelain") commands and low level ("plumbing") commands.

High-level commands (porcelain)

We separate the porcelain commands into the main commands and some ancillary user utilities.

1. Main porcelain commands

Section G.3.2, “git-add(1)”

Add file contents to the index.

Section G.3.3, “git-am(1)”

Apply a series of patches from a mailbox.

Section G.3.7, “git-archive(1)”

Create an archive of files from a named tree.

Section G.3.8, “git-bisect(1)”

Use binary search to find the commit that introduced a bug.

Section G.3.10, “git-branch(1)”

List, create, or delete branches.

Section G.3.12, “git-bundle(1)”

Move objects and refs by archive.

Section G.3.19, “git-checkout(1)”

Switch branches or restore working tree files.

Section G.3.20, “git-cherry-pick(1)”

Apply the changes introduced by some existing commits.

Section G.3.22, “git-citool(1)”

Graphical alternative to git-commit.

Section G.3.23, “git-clean(1)”

Remove untracked files from the working tree.

Section G.3.24, “git-clone(1)”

Clone a repository into a new directory.

Section G.3.28, “git-commit(1)”

Record changes to the repository.

Section G.3.39, “git-describe(1)”

Give an object a human readable name based on an available ref.

Section G.3.44, “git-diff(1)”

Show changes between commits, commit and working tree, etc.

Section G.3.49, “git-fetch(1)”

Download objects and refs from another repository.

Section G.3.54, “git-format-patch(1)”

Prepare patches for e-mail submission.

Section G.3.58, “git-gc(1)”

Cleanup unnecessary files and optimize the local repository.

Section G.3.60, “git-grep(1)”

Print lines matching a pattern.

Section G.3.61, “git-gui(1)”

A portable graphical interface to Git.

Section G.3.71, “git-init(1)”

Create an empty Git repository or reinitialize an existing one.

Section G.3.74, “git-log(1)”

Show commit logs.

Section G.3.80, “git-maintenance(1)”

Run tasks to optimize Git repository data.

Section G.3.86, “git-merge(1)”

Join two or more development histories together.

Section G.3.91, “git-mv(1)”

Move or rename a file, a directory, or a symlink.

Section G.3.94, “git-notes(1)”

Add or inspect object notes.

Section G.3.102, “git-pull(1)”

Fetch from and integrate with another repository or a local branch.

Section G.3.103, “git-push(1)”

Update remote refs along with associated objects.

Section G.3.105, “git-range-diff(1)”

Compare two commit ranges (e.g. two versions of a branch).

Section G.3.107, “git-rebase(1)”

Reapply commits on top of another base tip.

Section G.3.118, “git-reset(1)”

Reset current HEAD to the specified state.

Section G.3.119, “git-restore(1)”

Restore working tree files.

Section G.3.122, “git-revert(1)”

Revert some existing commits.

Section G.3.123, “git-rm(1)”

Remove files from the working tree and from the index.

Section G.3.130, “git-shortlog(1)”

Summarize *git log* output.

Section G.3.134, “git-show(1)”

Show various types of objects.

Section G.3.135, “git-sparse-checkout(1)”

Reduce your working tree to a subset of tracked files.

Section G.3.137, “git-stash(1)”

Stash the changes in a dirty working directory away.

Section G.3.138, “git-status(1)”

Show the working tree status.

Section G.3.141, “git-submodule(1)”

Initialize, update or inspect submodules.

Section G.3.140, “git-switch(1)”

Switch branches.

Section G.3.144, “git-tag(1)”

Create, list, delete or verify a tag object signed with GPG.

Section G.3.159, “git-worktree(1)”

Manage multiple working trees.

Section G.4.8, “gitk(1)”

The Git repository browser.

Section G.3.161, “scalar(1)”

A tool for managing large Git repositories.

2. Ancillary Commands

Manipulators:

Section G.3.29, “git-config(1)”

Get and set repository or global options.

Section G.3.46, “git-fast-export(1)”

Git data exporter.

Section G.3.47, “git-fast-import(1)”

Backend for fast Git data importers.

Section G.3.50, “git-filter-branch(1)”

Rewrite branches.

Section G.3.88, “git-mergetool(1)”

Run merge conflict resolution tools to resolve merge conflicts.

Section G.3.98, “git-pack-refs(1)”

Pack heads and tags for efficient repository access.

Section G.3.101, “git-prune(1)”

Prune all unreachable objects from the object database.

Section G.3.109, “git-reflog(1)”

Manage reflog information.

Section G.3.112, “git-remote(1)”

Manage set of tracked repositories.

Section G.3.113, “git-repack(1)”

Pack unpacked objects in a repository.

Section G.3.114, “git-replace(1)”

Create, list, delete refs to replace objects.

Interrogators:

Section G.3.4, “git-annotate(1)”

Annotate file lines with commit information.

Section G.3.9, “git-blame(1)”

Show what revision and author last modified each line of a file.

Section G.3.11, “git-bugreport(1)”

Collect information for user to file a bug report.

Section G.3.30, “git-count-objects(1)”

Count unpacked number of objects and their disk consumption.

Section G.3.40, “git-diagnose(1)”

Generate a zip archive of diagnostic information.

Section G.3.45, “git-difftool(1)”

Show changes using common diff tools.

Section G.3.56, “git-fsck(1)”

Verifies the connectivity and validity of the objects in the database.

Section G.3.63, “git-help(1)”

Display help information about Git.

Section G.3.72, “git-instaweb(1)”

Instantly browse your working repository in gitweb.

Section G.3.85, “git-merge-tree(1)”

Perform merge without touching index or working tree.

Section G.3.117, “git-rerere(1)”

Reuse recorded resolution of conflicted merges.

Section G.3.131, “git-show-branch(1)”

Show branches and their commits.

Section G.3.153, “git-verify-commit(1)”

Check the GPG signature of commits.

Section G.3.155, “git-verify-tag(1)”

Check the GPG signature of tags.

Section G.3.156, “git-version(1)”

Display version information about Git.

Section G.3.158, “git-whatchanged(1)”

Show logs with differences each commit introduces.

Section G.4.16, “gitweb(1)”

Git web interface (web frontend to Git repositories).

3. Interacting with Others

These commands are to interact with foreign SCM and with other people via patch over e-mail.

Section G.3.6, “git-archimport(1)”

Import a GNU Arch repository into Git.

Section G.3.35, “git-cvsexportcommit(1)”

Export a single commit to a CVS checkout.

Section G.3.36, “git-cvsimport(1)”

Salvage your data out of another SCM people love to hate.

Section G.3.37, “git-cvsserver(1)”

A CVS server emulator for Git.

Section G.3.68, “git-imap-send(1)”

Send a collection of patches from stdin to an IMAP folder.

Section G.3.95, “git-p4(1)”

Import from and submit to Perforce repositories.

Section G.3.104, “git-quiltimport(1)”

Applies a quilt patchset onto the current branch.

Section G.3.116, “git-request-pull(1)”

Generates a summary of pending changes.

Section G.3.124, “git-send-email(1)”

Send a collection of patches as emails.

Section G.3.142, “git-svn(1)”

Bidirectional operation between a Subversion repository and Git.

4. Reset, restore and revert

There are three commands with similar names: *git reset*, *git restore* and *git revert*.

- Section G.3.122, “git-revert(1)” is about making a new commit that reverts the changes made by other commits.
- Section G.3.119, “git-restore(1)” is about restoring files in the working tree from either the index or another commit. This command does not update your branch. The command can also be used to restore files in the index from another commit.
- Section G.3.118, “git-reset(1)” is about updating your branch, moving the tip in order to add or remove commits from the branch. This operation changes the commit history.

  git reset can also be used to restore the index, overlapping with *git restore*.

Low-level commands (plumbing)

Although Git includes its own porcelain layer, its low-level commands are sufficient to support development of alternative porcelains. Developers of such porcelains might start by reading about Section G.3.147, “git-update-index(1)” and Section G.3.106, “git-read-tree(1)”.

The interface (input, output, set of options and the semantics) to these low-level commands are meant to be a lot more stable than Porcelain level commands, because these commands are primarily for scripted use. The interface to Porcelain commands on the other hand are subject to change in order to improve the end user experience.

The following description divides the low-level commands into commands that manipulate objects (in the repository, index, and working tree), commands that interrogate and compare objects, and commands that move objects and references between repositories.

1. Manipulation commands

Section G.3.5, “git-apply(1)”

Apply a patch to files and/or to the index.

Section G.3.18, “git-checkout-index(1)”

Copy files from the index to the working tree.

Section G.3.26, “git-commit-graph(1)”

Write and verify Git commit-graph files.

Section G.3.27, “git-commit-tree(1)”

Create a new commit object.

Section G.3.62, “git-hash-object(1)”

Compute object ID and optionally create an object from a file.

Section G.3.69, “git-index-pack(1)”

Build pack index file for an existing packed archive.

Section G.3.82, “git-merge-file(1)”

Run a three-way file merge.

Section G.3.83, “git-merge-index(1)”

Run a merge for files needing merging.

Section G.3.89, “git-mktag(1)”

Creates a tag object with extra validation.

Section G.3.90, “git-mktree(1)”

Build a tree-object from ls-tree formatted text.

Section G.3.92, “git-multi-pack-index(1)”

Write and verify multi-pack-indexes.

Section G.3.96, “git-pack-objects(1)”

Create a packed archive of objects.

Section G.3.100, “git-prune-packed(1)”

Remove extra objects that are already in pack files.

Section G.3.106, “git-read-tree(1)”

Reads tree information into the index.

Section G.3.115, “git-replay(1)”

EXPERIMENTAL: Replay commits on a new base, works with bare repos too.

Section G.3.143, “git-symbolic-ref(1)”

Read, modify and delete symbolic refs.

Section G.3.146, “git-unpack-objects(1)”

Unpack objects from a packed archive.

Section G.3.147, “git-update-index(1)”

Register file contents in the working tree to the index.

Section G.3.148, “git-update-ref(1)”

Update the object name stored in a ref safely.

Section G.3.160, “git-write-tree(1)”

Create a tree object from the current index.

2. Interrogation commands

Section G.3.13, “git-cat-file(1)”

Provide contents or details of repository objects.

Section G.3.21, “git-cherry(1)”

Find commits yet to be applied to upstream.

Section G.3.41, “git-diff-files(1)”

Compares files in the working tree and the index.

Section G.3.42, “git-diff-index(1)”

Compare a tree to the working tree or index.

Section G.3.43, “git-diff-tree(1)”

Compares the content and mode of blobs found via two tree objects.

Section G.3.52, “git-for-each-ref(1)”

Output information on each ref.

Section G.3.53, “git-for-each-repo(1)”

Run a Git command on a list of repositories.

Section G.3.59, “git-get-tar-commit-id(1)”

Extract commit ID from an archive created using git-archive.

Section G.3.75, “git-ls-files(1)”

Show information about files in the index and the working tree.

Section G.3.76, “git-ls-remote(1)”

List references in a remote repository.

Section G.3.77, “git-ls-tree(1)”

List the contents of a tree object.

Section G.3.81, “git-merge-base(1)”

Find as good common ancestors as possible for a merge.

Section G.3.93, “git-name-rev(1)”

Find symbolic names for given revs.

Section G.3.97, “git-pack-redundant(1)”

Find redundant pack files.

Section G.3.120, “git-rev-list(1)”

Lists commit objects in reverse chronological order.

Section G.3.121, “git-rev-parse(1)”

Pick out and massage parameters.

Section G.3.132, “git-show-index(1)”

Show packed archive index.

Section G.3.133, “git-show-ref(1)”

List references in a local repository.

Section G.3.145, “git-unpack-file(1)”

Creates a temporary file with a blob's contents.

Section G.3.152, “git-var(1)”

Show a Git logical variable.

Section G.3.154, “git-verify-pack(1)”

Validate packed Git archive files.

In general, the interrogate commands do not touch the files in the working tree.

3. Syncing repositories

Section G.3.38, “git-daemon(1)”

A really simple server for Git repositories.

Section G.3.48, “git-fetch-pack(1)”

Receive missing objects from another repository.

Section G.3.65, “git-http-backend(1)”

Server side implementation of Git over HTTP.

Section G.3.125, “git-send-pack(1)”

Push objects over Git protocol to another repository.

Section G.3.149, “git-update-server-info(1)”

Update auxiliary info file to help dumb servers.

The following are helper commands used by the above; end users typically do not use them directly.

Section G.3.66, “git-http-fetch(1)”

Download from a remote Git repository via HTTP.

Section G.3.67, “git-http-push(1)”

Push objects over HTTP/DAV to another repository.

Section G.3.108, “git-receive-pack(1)”

Receive what is pushed into the repository.

Section G.3.129, “git-shell(1)”

Restricted login shell for Git-only SSH access.

Section G.3.150, “git-upload-archive(1)”

Send archive back to git-archive.

Section G.3.151, “git-upload-pack(1)”

Send objects packed back to git-fetch-pack.

4. Internal helper commands

These are internal helper commands used by other commands; end users typically do not use them directly.

Section G.3.14, “git-check-attr(1)”

Display gitattributes information.

Section G.3.15, “git-check-ignore(1)”

Debug gitignore / exclude files.

Section G.3.16, “git-check-mailmap(1)”

Show canonical names and email addresses of contacts.

Section G.3.17, “git-check-ref-format(1)”

Ensures that a reference name is well formed.

Section G.3.25, “git-column(1)”

Display data in columns.

Section G.3.31, “git-credential(1)”

Retrieve and store user credentials.

Section G.3.33, “git-credential-cache(1)”

Helper to temporarily store passwords in memory.

Section G.3.34, “git-credential-store(1)”

Helper to store credentials on disk.

Section G.3.51, “git-fmt-merge-msg(1)”

Produce a merge commit message.

Section G.3.64, “git-hook(1)”

Run git hooks.

Section G.3.73, “git-interpret-trailers(1)”

Add or parse structured information in commit messages.

Section G.3.78, “git-mailinfo(1)”

Extracts patch and authorship from a single e-mail message.

Section G.3.79, “git-mailsplit(1)”

Simple UNIX mbox splitter program.

Section G.3.84, “git-merge-one-file(1)”

The standard helper program to use with git-merge-index.

Section G.3.99, “git-patch-id(1)”

Compute unique ID for a patch.

Section G.3.127, “git-sh-i18n(1)”

Git's i18n setup code for shell scripts.

Section G.3.128, “git-sh-setup(1)”

Common Git shell script setup code.

Section G.3.139, “git-stripspace(1)”

Remove unnecessary whitespace.

Guides

The following documentation pages are guides about Git concepts.

Section G.2.3, “gitcore-tutorial(7)”

A Git core tutorial for developers.

Section G.4.3, “gitcredentials(7)”

Providing usernames and passwords to Git.

Section G.2.4, “gitcvs-migration(7)”

Git for CVS users.

Section G.4.4, “gitdiffcore(7)”

Tweaking diff output.

Section G.2.5, “giteveryday(7)”

A useful minimum set of commands for Everyday Git.

Section G.4.6, “gitfaq(7)”

Frequently asked questions about using Git.

Section G.4.19, “gitglossary(7)”

A Git Glossary.

Section G.4.11, “gitnamespaces(7)”

Git namespaces.

Section G.4.12, “gitremote-helpers(7)”

Helper programs to interact with remote repositories.

Section G.4.15, “gitsubmodules(7)”

Mounting one repository inside another.

Section G.2.1, “gittutorial(7)”

A tutorial introduction to Git.

Section G.2.2, “gittutorial-2(7)”

A tutorial introduction to Git: part two.

Section G.4.18, “gitworkflows(7)”

An overview of recommended workflows with Git.

Repository, command and file interfaces

This documentation discusses repository and command interfaces which users are expected to interact with directly. See *--user-formats* in Section G.3.63, “git-help(1)” for more details on the criteria.

Section G.4.2, “gitattributes(5)”

Defining attributes per path.

Section G.4.1, “gitcli(7)”

Git command-line interface and conventions.

Section G.4.7, “githooks(5)”

Hooks used by Git.

Section G.4.5, “gitignore(5)”

Specifies intentionally untracked files to ignore.

Section G.4.9, “gitmailmap(5)”

Map author/committer names and/or E-Mail addresses.

Section G.4.10, “gitmodules(5)”

Defining submodule properties.

Section G.4.13, “gitrepository-layout(5)”

Git Repository Layout.

Section G.4.14, “gitrevisions(7)”

Specifying revisions and ranges for Git.

File formats, protocols and other developer interfaces

This documentation discusses file formats, over-the-wire protocols and other git developer interfaces. See *--developer-interfaces* in Section G.3.63, “git-help(1)”.

Section G.5.1, “gitformat-bundle(5)”

The bundle file format.

Section G.5.2, “gitformat-chunk(5)”

Chunk-based file formats.

Section G.5.3, “gitformat-commit-graph(5)”

Git commit-graph format.

Section G.5.4, “gitformat-index(5)”

Git index format.

Section G.5.5, “gitformat-pack(5)”

Git pack format.

Section G.5.6, “gitformat-signature(5)”

Git cryptographic signature formats.

Section G.5.7, “gitprotocol-capabilities(5)”

Protocol v0 and v1 capabilities.

Section G.5.8, “gitprotocol-common(5)”

Things common to various protocols.

Section G.5.9, “gitprotocol-http(5)”

Git HTTP-based protocols.

Section G.5.10, “gitprotocol-pack(5)”

How packs are transferred over-the-wire.

Section G.5.11, “gitprotocol-v2(5)”

Git Wire Protocol, Version 2.

Configuration Mechanism

Git uses a simple text format to store customizations that are per repository and are per user. Such a configuration file may look like this:

```
#
# A '#' or ';' character indicates a comment.
#

; core variables
[core]
        ; Don't trust file modes
        filemode = false

; user identity
[user]
        name = "Junio C Hamano"
        email = "gitster@pobox.com"
```

Various commands read from the configuration file and adjust their operation accordingly. See Section G.3.29, “git-config(1)” for a list and more details about the configuration mechanism.

Identifier Terminology

<object>

Indicates the object name for any type of object.

<blob>

Indicates a blob object name.

<tree>

Indicates a tree object name.

<commit>

Indicates a commit object name.

<tree-ish>

Indicates a tree, commit or tag object name. A command that takes a <tree-ish> argument ultimately wants to operate on a <tree> object but automatically dereferences <commit> and <tag> objects that point at a <tree>.

<commit-ish>

Indicates a commit or tag object name. A command that takes a <commit-ish> argument ultimately wants to operate on a <commit> object but automatically dereferences <tag> objects that point at a <commit>.

<type>

Indicates that an object type is required. Currently one of: *blob*, *tree*, *commit*, or *tag*.

<file>

Indicates a filename - almost always relative to the root of the tree structure *GIT_INDEX_FILE* describes.

Symbolic Identifiers

Any Git command accepting any <object> can also use the following symbolic notation:

HEAD

indicates the head of the current branch.

<tag>

a valid tag *name* (i.e. a *refs/tags/<tag>* reference).

<head>

a valid head *name* (i.e. a *refs/heads/<head>* reference).

For a more complete list of ways to spell object names, see "SPECIFYING REVISIONS" section in Section G.4.14, “gitrevisions(7)”.

File/Directory Structure

Please see the Section G.4.13, “gitrepository-layout(5)” document.

Read Section G.4.7, “githooks(5)” for more details about each hook.

Higher level SCMs may provide and manage additional information in the *$GIT_DIR*.

Terminology

Please see Section G.4.19, “gitglossary(7)”.

Environment Variables

Various Git commands pay attention to environment variables and change their behavior. The environment variables marked as "Boolean" take their values the same way as Boolean valued configuration variables, e.g. "true", "yes", "on" and positive numbers are taken as "yes".

Here are the variables:

1. The Git Repository

These environment variables apply to *all* core Git commands. Nb: it is worth noting that they may be used/overridden by SCMS sitting above Git so take care if using a foreign front-end.

GIT_INDEX_FILE

This environment variable specifies an alternate index file. If not specified, the default of *$GIT_DIR/index* is used.

GIT_INDEX_VERSION

This environment variable specifies what index version is used when writing the index file out. It won't affect existing index files. By default index file version 2 or 3 is used. See Section G.3.147, “git-update-index(1)” for more information.

GIT_OBJECT_DIRECTORY

If the object storage directory is specified via this environment variable then the sha1 directories are created underneath - otherwise the default *$GIT_DIR/objects* directory is used.

GIT_ALTERNATE_OBJECT_DIRECTORIES

Due to the immutable nature of Git objects, old objects can be archived into shared, read-only directories. This variable specifies a ":" separated (on Windows ";" separated) list of Git object directories which can be used to search for Git objects. New objects will not be written to these directories.

Entries that begin with *"* (double-quote) will be interpreted as C-style quoted paths, removing leading and trailing double-quotes and respecting backslash escapes. E.g., the value *"path-with-\"-and-:-in-it":vanilla-path* has two paths: *path-with-"-and-:-in-it* and *vanilla-path*.

GIT_DIR

If the *GIT_DIR* environment variable is set then it specifies a path to use instead of the default *.git* for the base of the repository. The *--git-dir* command-line option also sets this value.

GIT_WORK_TREE

Set the path to the root of the working tree. This can also be controlled by the *--work-tree* command-line option and the core.worktree configuration variable.

GIT_NAMESPACE

Set the Git namespace; see Section G.4.11, “gitnamespaces(7)” for details. The *--namespace* command-line option also sets this value.

GIT_CEILING_DIRECTORIES

This should be a colon-separated list of absolute paths. If set, it is a list of directories that Git should not chdir up into while looking for a repository directory (useful for excluding slow-loading network directories). It will not exclude the current working directory or a GIT_DIR set on the command line or in the environment. Nor-

mally, Git has to read the entries in this list and resolve any symlink that might be present in order to compare them with the current directory. However, if even this access is slow, you can add an empty entry to the list to tell Git that the subsequent entries are not symlinks and needn't be resolved; e.g., *GIT_CEILING_DIRECTORIES=/maybe/symlink::/very/slow/non/symlink*.

GIT_DISCOVERY_ACROSS_FILESYSTEM

When run in a directory that does not have ".git" repository directory, Git tries to find such a directory in the parent directories to find the top of the working tree, but by default it does not cross filesystem boundaries. This Boolean environment variable can be set to true to tell Git not to stop at filesystem boundaries. Like *GIT_CEILING_DIRECTORIES*, this will not affect an explicit repository directory set via *GIT_DIR* or on the command line.

GIT_COMMON_DIR

If this variable is set to a path, non-worktree files that are normally in $GIT_DIR will be taken from this path instead. Worktree-specific files such as HEAD or index are taken from $GIT_DIR. See Section G.4.13, “gitrepository-layout(5)” and Section G.3.159, “git-worktree(1)” for details. This variable has lower precedence than other path variables such as GIT_INDEX_FILE, GIT_OBJECT_DIRECTORY…

GIT_DEFAULT_HASH

If this variable is set, the default hash algorithm for new repositories will be set to this value. This value is ignored when cloning and the setting of the remote repository is always used. The default is "sha1". See *--object-format* in Section G.3.71, “git-init(1)”.

GIT_DEFAULT_REF_FORMAT

If this variable is set, the default reference backend format for new repositories will be set to this value. The default is "files". See *--ref-format* in Section G.3.71, “git-init(1)”.

2. Git Commits

GIT_AUTHOR_NAME

The human-readable name used in the author identity when creating commit or tag objects, or when writing reflogs. Overrides the *user.name* and *author.name* configuration settings.

GIT_AUTHOR_EMAIL

The email address used in the author identity when creating commit or tag objects, or when writing reflogs. Overrides the *user.email* and *author.email* configuration settings.

GIT_AUTHOR_DATE

The date used for the author identity when creating commit or tag objects, or when writing reflogs. See Section G.3.28, “git-commit(1)” for valid formats.

GIT_COMMITTER_NAME

The human-readable name used in the committer identity when creating commit or tag objects, or when writing reflogs. Overrides the *user.name* and *committer.name* configuration settings.

GIT_COMMITTER_EMAIL

The email address used in the author identity when creating commit or tag objects, or when writing reflogs. Overrides the *user.email* and *committer.email* configuration settings.

GIT_COMMITTER_DATE

The date used for the committer identity when creating commit or tag objects, or when writing reflogs. See Section G.3.28, “git-commit(1)” for valid formats.

EMAIL

The email address used in the author and committer identities if no other relevant environment variable or configuration setting has been set.

3. Git Diffs

GIT_DIFF_OPTS

Only valid setting is "--unified=??" or "-u??" to set the number of context lines shown when a unified diff is created. This takes precedence over any "-U" or "--unified" option value passed on the Git diff command line.

GIT_EXTERNAL_DIFF

When the environment variable *GIT_EXTERNAL_DIFF* is set, the program named by it is called to generate diffs, and Git does not use its builtin diff machinery. For a path that is added, removed, or modified, *GIT_EXTERNAL_DIFF* is called with 7 parameters:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

where:

<old|new>-file

are files GIT_EXTERNAL_DIFF can use to read the contents of <old|new>,

<old|new>-hex

are the 40-hexdigit SHA-1 hashes,

<old|new>-mode

are the octal representation of the file modes.

The file parameters can point at the user's working file (e.g. *new-file* in "git-diff-files"), */dev/null* (e.g. *old-file* when a new file is added), or a temporary file (e.g. *old-file* in the index). *GIT_EXTERNAL_DIFF* should not worry about unlinking the temporary file -- it is removed when *GIT_EXTERNAL_DIFF* exits.

For a path that is unmerged, *GIT_EXTERNAL_DIFF* is called with 1 parameter, <path>.

For each path *GIT_EXTERNAL_DIFF* is called, two environment variables, *GIT_DIFF_PATH_COUNTER* and *GIT_DIFF_PATH_TOTAL* are set.

GIT_DIFF_PATH_COUNTER

A 1-based counter incremented by one for every path.

GIT_DIFF_PATH_TOTAL

The total number of paths.

4. other

GIT_MERGE_VERBOSITY

A number controlling the amount of output shown by the recursive merge strategy. Overrides merge.verbosity. See Section G.3.86, “git-merge(1)”

GIT_PAGER

This environment variable overrides *$PAGER*. If it is set to an empty string or to the value "cat", Git will not launch a pager. See also the *core.pager* option in Section G.3.29, “git-config(1)”.

GIT_PROGRESS_DELAY

A number controlling how many seconds to delay before showing optional progress indicators. Defaults to 2.

GIT_EDITOR

This environment variable overrides *$EDITOR* and *$VISUAL*. It is used by several Git commands when, on interactive mode, an editor is to be launched. See also Section G.3.152, "git-var(1)" and the *core.editor* option in Section G.3.29, "git-config(1)".

GIT_SEQUENCE_EDITOR

This environment variable overrides the configured Git editor when editing the todo list of an interactive rebase. See also Section G.3.107, "git-rebase(1)" and the *sequence.editor* option in Section G.3.29, "git-config(1)".

GIT_SSH , *GIT_SSH_COMMAND*

If either of these environment variables is set then *git fetch* and *git push* will use the specified command instead of *ssh* when they need to connect to a remote system. The command-line parameters passed to the configured command are determined by the ssh variant. See *ssh.variant* option in Section G.3.29, "git-config(1)" for details.

$GIT_SSH_COMMAND takes precedence over *$GIT_SSH*, and is interpreted by the shell, which allows additional arguments to be included. *$GIT_SSH* on the other hand must be just the path to a program (which can be a wrapper shell script, if additional arguments are needed).

Usually it is easier to configure any desired options through your personal *.ssh/config* file. Please consult your ssh documentation for further details.

GIT_SSH_VARIANT

If this environment variable is set, it overrides Git's autodetection whether *GIT_SSH*/*GIT_SSH_COMMAND*/ *core.sshCommand* refer to OpenSSH, plink or tortoiseplink. This variable overrides the config setting *ssh.variant* that serves the same purpose.

GIT_SSL_NO_VERIFY

Setting and exporting this environment variable to any value tells Git not to verify the SSL certificate when fetching or pushing over HTTPS.

GIT_ATTR_SOURCE

Sets the treeish that gitattributes will be read from.

GIT_ASKPASS

If this environment variable is set, then Git commands which need to acquire passwords or passphrases (e.g. for HTTP or IMAP authentication) will call this program with a suitable prompt as command-line argument and read the password from its STDOUT. See also the *core.askPass* option in Section G.3.29, "git-config(1)".

GIT_TERMINAL_PROMPT

If this Boolean environment variable is set to false, git will not prompt on the terminal (e.g., when asking for HTTP authentication).

GIT_CONFIG_GLOBAL , *GIT_CONFIG_SYSTEM*

Take the configuration from the given files instead from global or system-level configuration files. If *GIT_CONFIG_SYSTEM* is set, the system config file defined at build time (usually */etc/gitconfig*) will not be read. Likewise, if *GIT_CONFIG_GLOBAL* is set, neither *$HOME/.gitconfig* nor *$XDG_CONFIG_HOME/git/config* will be read. Can be set to */dev/null* to skip reading configuration files of the respective level.

GIT_CONFIG_NOSYSTEM

Whether to skip reading settings from the system-wide *$(prefix)/etc/gitconfig* file. This Boolean environment variable can be used along with *$HOME* and *$XDG_CONFIG_HOME* to create a predictable environment for a picky script, or you can set it to true to temporarily avoid using a buggy */etc/gitconfig* file while waiting for someone with sufficient permissions to fix it.

GIT_FLUSH

If this Boolean environment variable is set to true, then commands such as *git blame* (in incremental mode), *git rev-list*, *git log*, *git check-attr* and *git check-ignore* will force a flush of the output stream after each record have been flushed. If this variable is set to false, the output of these commands will be done using completely buffered I/O. If this environment variable is not set, Git will choose buffered or record-oriented flushing based on whether stdout appears to be redirected to a file or not.

GIT_TRACE

Enables general trace messages, e.g. alias expansion, built-in command execution and external command execution.

If this variable is set to "1", "2" or "true" (comparison is case insensitive), trace messages will be printed to stderr.

If the variable is set to an integer value greater than 2 and lower than 10 (strictly) then Git will interpret this value as an open file descriptor and will try to write the trace messages into this file descriptor.

Alternatively, if the variable is set to an absolute path (starting with a / character), Git will interpret this as a file path and will try to append the trace messages to it.

Unsetting the variable, or setting it to empty, "0" or "false" (case insensitive) disables trace messages.

GIT_TRACE_FSMONITOR

Enables trace messages for the filesystem monitor extension. See *GIT_TRACE* for available trace output options.

GIT_TRACE_PACK_ACCESS

Enables trace messages for all accesses to any packs. For each access, the pack file name and an offset in the pack is recorded. This may be helpful for troubleshooting some pack-related performance problems. See *GIT_TRACE* for available trace output options.

GIT_TRACE_PACKET

Enables trace messages for all packets coming in or out of a given program. This can help with debugging object negotiation or other protocol issues. Tracing is turned off at a packet starting with "PACK" (but see *GIT_TRACE_PACKFILE* below). See *GIT_TRACE* for available trace output options.

GIT_TRACE_PACKFILE

Enables tracing of packfiles sent or received by a given program. Unlike other trace output, this trace is verbatim: no headers, and no quoting of binary data. You almost certainly want to direct into a file (e.g., *GIT_TRACE_PACKFILE=/tmp/my.pack*) rather than displaying it on the terminal or mixing it with other trace output.

Note that this is currently only implemented for the client side of clones and fetches.

GIT_TRACE_PERFORMANCE

Enables performance related trace messages, e.g. total execution time of each Git command. See *GIT_TRACE* for available trace output options.

GIT_TRACE_REFS

Enables trace messages for operations on the ref database. See *GIT_TRACE* for available trace output options.

GIT_TRACE_SETUP

Enables trace messages printing the .git, working tree and current working directory after Git has completed its setup phase. See *GIT_TRACE* for available trace output options.

GIT_TRACE_SHALLOW

Enables trace messages that can help debugging fetching / cloning of shallow repositories. See *GIT_TRACE* for available trace output options.

GIT_TRACE_CURL

Enables a curl full trace dump of all incoming and outgoing data, including descriptive information, of the git transport protocol. This is similar to doing curl *--trace-ascii* on the command line. See *GIT_TRACE* for available trace output options.

GIT_TRACE_CURL_NO_DATA

When a curl trace is enabled (see *GIT_TRACE_CURL* above), do not dump data (that is, only dump info lines and headers).

GIT_TRACE2

Enables more detailed trace messages from the "trace2" library. Output from *GIT_TRACE2* is a simple text-based format for human readability.

If this variable is set to "1", "2" or "true" (comparison is case insensitive), trace messages will be printed to stderr.

If the variable is set to an integer value greater than 2 and lower than 10 (strictly) then Git will interpret this value as an open file descriptor and will try to write the trace messages into this file descriptor.

Alternatively, if the variable is set to an absolute path (starting with a / character), Git will interpret this as a file path and will try to append the trace messages to it. If the path already exists and is a directory, the trace messages will be written to files (one per process) in that directory, named according to the last component of the SID and an optional counter (to avoid filename collisions).

In addition, if the variable is set to *af_unix:[<socket-type>:]<absolute-pathname>*, Git will try to open the path as a Unix Domain Socket. The socket type can be either *stream* or *dgram*.

Unsetting the variable, or setting it to empty, "0" or "false" (case insensitive) disables trace messages.

See *Trace2 documentation* [https://www.kernel.org/pub/software/scm/git/docs/technical/api-trace2.html] for full details.

GIT_TRACE2_EVENT

This setting writes a JSON-based format that is suited for machine interpretation. See *GIT_TRACE2* for available trace output options and *Trace2 documentation* [https://www.kernel.org/pub/software/scm/git/docs/technical/api-trace2.html] for full details.

GIT_TRACE2_PERF

In addition to the text-based messages available in *GIT_TRACE2*, this setting writes a column-based format for understanding nesting regions. See *GIT_TRACE2* for available trace output options and *Trace2 documentation* [https://www.kernel.org/pub/software/scm/git/docs/technical/api-trace2.html] for full details.

GIT_TRACE_REDACT

By default, when tracing is activated, Git redacts the values of cookies, the "Authorization:" header, the "Proxy-Authorization:" header and packfile URIs. Set this Boolean environment variable to false to prevent this redaction.

GIT_NO_REPLACE_OBJECTS

Setting and exporting this environment variable tells Git to ignore replacement refs and do not replace Git objects.

GIT_LITERAL_PATHSPECS

Setting this Boolean environment variable to true will cause Git to treat all pathspecs literally, rather than as glob patterns. For example, running *GIT_LITERAL_PATHSPECS=1 git log -- '*.c'* will search for commits that touch the path **.c*, not any paths that the glob **.c* matches. You might want this if you are feeding literal paths to Git (e.g., paths previously given to you by *git ls-tree*, *--raw* diff output, etc).

GIT_GLOB_PATHSPECS

Setting this Boolean environment variable to true will cause Git to treat all pathspecs as glob patterns (aka "glob" magic).

GIT_NOGLOB_PATHSPECS

Setting this Boolean environment variable to true will cause Git to treat all pathspecs as literal (aka "literal" magic).

GIT_ICASE_PATHSPECS

Setting this Boolean environment variable to true will cause Git to treat all pathspecs as case-insensitive.

GIT_NO_LAZY_FETCH

Setting this Boolean environment variable to true tells Git not to lazily fetch missing objects from the promisor remote on demand.

GIT_REFLOG_ACTION

When a ref is updated, reflog entries are created to keep track of the reason why the ref was updated (which is typically the name of the high-level command that updated the ref), in addition to the old and new values of the ref. A scripted Porcelain command can use set_reflog_action helper function in *git-sh-setup* to set its name to this variable when it is invoked as the top level command by the end user, to be recorded in the body of the reflog.

GIT_REF_PARANOIA

If this Boolean environment variable is set to false, ignore broken or badly named refs when iterating over lists of refs. Normally Git will try to include any such refs, which may cause some operations to fail. This is usually preferable, as potentially destructive operations (e.g., Section G.3.101, “git-prune(1)”) are better off aborting rather than ignoring broken refs (and thus considering the history they point to as not worth saving). The default value is *1* (i.e., be paranoid about detecting and aborting all operations). You should not normally need to set this to *0*, but it may be useful when trying to salvage data from a corrupted repository.

GIT_COMMIT_GRAPH_PARANOIA

When loading a commit object from the commit-graph, Git performs an existence check on the object in the object database. This is done to avoid issues with stale commit-graphs that contain references to already-deleted commits, but comes with a performance penalty.

The default is "false", which disables the aforementioned behavior. Setting this to "true" enables the existence check so that stale commits will never be returned from the commit-graph at the cost of performance.

GIT_ALLOW_PROTOCOL

If set to a colon-separated list of protocols, behave as if *protocol.allow* is set to *never*, and each of the listed protocols has *protocol.<name>.allow* set to *always* (overriding any existing configuration). See the description of *protocol.allow* in Section G.3.29, "git-config(1)" for more details.

GIT_PROTOCOL_FROM_USER

Set this Boolean environment variable to false to prevent protocols used by fetch/push/clone which are configured to the *user* state. This is useful to restrict recursive submodule initialization from an untrusted repository or for programs which feed potentially-untrusted URLS to git commands. See Section G.3.29, "git-config(1)" for more details.

GIT_PROTOCOL

For internal use only. Used in handshaking the wire protocol. Contains a colon *:* separated list of keys with optional values *<key>[=<value>]*. Presence of unknown keys and values must be ignored.

Note that servers may need to be configured to allow this variable to pass over some transports. It will be propagated automatically when accessing local repositories (i.e., *file://* or a filesystem path), as well as over the *git://* protocol. For git-over-http, it should work automatically in most configurations, but see the discussion in Section G.3.65, "git-http-backend(1)". For git-over-ssh, the ssh server may need to be configured to allow clients to pass this variable (e.g., by using *AcceptEnv GIT_PROTOCOL* with OpenSSH).

This configuration is optional. If the variable is not propagated, then clients will fall back to the original "v0" protocol (but may miss out on some performance improvements or features). This variable currently only affects clones and fetches; it is not yet used for pushes (but may be in the future).

GIT_OPTIONAL_LOCKS

If this Boolean environment variable is set to false, Git will complete any requested operation without performing any optional sub-operations that require taking a lock. For example, this will prevent *git status* from refreshing the index as a side effect. This is useful for processes running in the background which do not want to cause lock contention with other operations on the repository. Defaults to *1*.

GIT_REDIRECT_STDIN , *GIT_REDIRECT_STDOUT* , *GIT_REDIRECT_STDERR*

Windows-only: allow redirecting the standard input/output/error handles to paths specified by the environment variables. This is particularly useful in multi-threaded applications where the canonical way to pass standard handles via *CreateProcess()* is not an option because it would require the handles to be marked inheritable (and consequently **every** spawned process would inherit them, possibly blocking regular Git operations). The primary intended use case is to use named pipes for communication (e.g. *\\.\pipe\my-git-stdin-123*).

Two special values are supported: *off* will simply close the corresponding standard handle, and if *GIT_REDIRECT_STDERR* is *2>&1*, standard error will be redirected to the same handle as standard output.

GIT_PRINT_SHA1_ELLIPSIS (deprecated)

If set to *yes*, print an ellipsis following an (abbreviated) SHA-1 value. This affects indications of detached HEADs (Section G.3.19, "git-checkout(1)") and the raw diff output (Section G.3.44, "git-diff(1)"). Printing an ellipsis in the cases mentioned is no longer considered adequate and support for it is likely to be removed in the foreseeable future (along with the variable).

Discussion

More detail on the following is available from the *Git concepts chapter of the user-manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html#git-concepts] and Section G.2.3, "gitcore-tutorial(7)".

A Git project normally consists of a working directory with a ".git" subdirectory at the top level. The .git directory contains, among other things, a compressed object database representing the complete history of the project, an "index" file which links that history to the current contents of the working tree, and named pointers into that history such as tags and branch heads.

The object database contains objects of three main types: blobs, which hold file data; trees, which point to blobs and other trees to build up directory hierarchies; and commits, which each reference a single tree and some number of parent commits.

The commit, equivalent to what other systems call a "changeset" or "version", represents a step in the project's history, and each parent represents an immediately preceding step. Commits with more than one parent represent merges of independent lines of development.

All objects are named by the SHA-1 hash of their contents, normally written as a string of 40 hex digits. Such names are globally unique. The entire history leading up to a commit can be vouched for by signing just that commit. A fourth object type, the tag, is provided for this purpose.

When first created, objects are stored in individual files, but for efficiency may later be compressed together into "pack files".

Named pointers called refs mark interesting points in history. A ref may contain the SHA-1 name of an object or the name of another ref (the latter is called a "symbolic ref"). Refs with names beginning *refs/head/* contain the SHA-1 name of the most recent commit (or "head") of a branch under development. SHA-1 names of tags of interest are stored under *refs/tags/*. A symbolic ref named *HEAD* contains the name of the currently checked-out branch.

The index file is initialized with a list of all paths and, for each path, a blob object and a set of attributes. The blob object represents the contents of the file as of the head of the current branch. The attributes (last modified time, size, etc.) are taken from the corresponding file in the working tree. Subsequent changes to the working tree can be found by comparing these attributes. The index may be updated with new content, and new commits may be created from the content stored in the index.

The index is also capable of storing multiple entries (called "stages") for a given pathname. These stages are used to hold the various unmerged version of a file when a merge is in progress.

SECURITY

Some configuration options and hook files may cause Git to run arbitrary shell commands. Because configuration and hooks are not copied using *git clone*, it is generally safe to clone remote repositories with untrusted content, inspect them with *git log*, and so on.

However, it is not safe to run Git commands in a *.git* directory (or the working tree that surrounds it) when that *.git* directory itself comes from an untrusted source. The commands in its config and hooks are executed in the usual way.

By default, Git will refuse to run when the repository is owned by someone other than the user running the command. See the entry for *safe.directory* in Section G.3.29, “git-config(1)”. While this can help protect you in a multi-user environment, note that you can also acquire untrusted repositories that are owned by you (for example, if you extract a zip file or tarball from an untrusted source). In such cases, you'd need to "sanitize" the untrusted repository first.

If you have an untrusted *.git* directory, you should first clone it with *git clone --no-local* to obtain a clean copy. Git does restrict the set of options and hooks that will be run by *upload-pack*, which handles the server side of a clone or fetch, but beware that the surface area for attack against *upload-pack* is large, so this does carry some risk. The safest thing is to serve the repository as an unprivileged user (either via Section G.3.38, “git-daemon(1)”, ssh, or using other tools to change user ids). See the discussion in the *SECURITY* section of Section G.3.151, “git-upload-pack(1)”.

FURTHER DOCUMENTATION

See the references in the "description" section to get started using Git. The following is probably more detail than necessary for a first-time user.

The *Git concepts chapter of the user-manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html#git-concepts] and Section G.2.3, “gitcore-tutorial(7)” both provide introductions to the underlying Git architecture.

See Section G.4.18, “gitworkflows(7)” for an overview of recommended workflows.

See also the *howto* [https://www.kernel.org/pub/software/scm/git/docs/howto-index.html] documents for some useful examples.

The internals are documented in the *Git API documentation* [https://www.kernel.org/pub/software/scm/git/docs/technical/api-index.html].

Users migrating from CVS may also want to read Section G.2.4, “gitcvs-migration(7)”.

Authors

Git was started by Linus Torvalds, and is currently maintained by Junio C Hamano. Numerous contributions have come from the Git mailing list <git@vger.kernel.org [mailto:git@vger.kernel.org]>. https://openhub.net/p/git/contributors/summary gives you a more complete list of contributors.

If you have a clone of git.git itself, the output of Section G.3.130, “git-shortlog(1)” and Section G.3.9, “git-blame(1)” can show you the authors for specific parts of the project.

Reporting Bugs

Report bugs to the Git mailing list <git@vger.kernel.org [mailto:git@vger.kernel.org]> where the development and maintenance is primarily done. You do not have to be subscribed to the list to send a message there. See the list archive at https://lore.kernel.org/git for previous bug reports and other discussions.

Issues which are security relevant should be disclosed privately to the Git Security mailing list <git-security@googlegroups.com [mailto:git-security@googlegroups.com]>.

SEE ALSO

Section G.2.1, “gittutorial(7)”, Section G.2.2, “gittutorial-2(7)”, Section G.2.5, “giteveryday(7)”, Section G.2.4, “gitcvs-migration(7)”, Section G.4.19, “gitglossary(7)”, Section G.2.3, “gitcore-tutorial(7)”, Section G.4.1, “gitcli(7)”, *The Git User's Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html], Section G.4.18, “gitworkflows(7)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.2. git-add(1)

2024-07-02

Revision History

2024-07-02

NAME

git-add - Add file contents to the index

SYNOPSIS

git add [--verbose | -v] [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]
[--edit | -e] [--[no-]all | -A | --[no-]ignore-removal | [--update | -u]] [--sparse]
[--intent-to-add | -N] [--refresh] [--ignore-errors] [--ignore-missing] [--renormalize]
[--chmod=(+|-)x] [--pathspec-from-file=<file> [--pathspec-file-nul]]
[--] [<pathspec>…]

DESCRIPTION

This command updates the index using the current content found in the working tree, to prepare the content staged for the next commit. It typically adds the current content of existing paths as a whole, but with some options it

can also be used to add content with only part of the changes made to the working tree files applied, or remove paths that do not exist in the working tree anymore.

The "index" holds a snapshot of the content of the working tree, and it is this snapshot that is taken as the contents of the next commit. Thus after making any changes to the working tree, and before running the commit command, you must use the *add* command to add any new or modified files to the index.

This command can be performed multiple times before a commit. It only adds the content of the specified file(s) at the time the add command is run; if you want subsequent changes included in the next commit, then you must run *git add* again to add the new content to the index.

The *git status* command can be used to obtain a summary of which files have changes that are staged for the next commit.

The *git add* command will not add ignored files by default. If any ignored files were explicitly specified on the command line, *git add* will fail with a list of ignored files. Ignored files reached by directory recursion or filename globbing performed by Git (quote your globs before the shell) will be silently ignored. The *git add* command can be used to add ignored files with the *-f* (force) option.

Please see Section G.3.28, “git-commit(1)” for alternative ways to add content to a commit.

OPTIONS

<pathspec>…

Files to add content from. Fileglobs (e.g. **.c*) can be given to add all matching files. Also a leading directory name (e.g. *dir* to add *dir/file1* and *dir/file2*) can be given to update the index to match the current state of the directory as a whole (e.g. specifying *dir* will record not just a file *dir/file1* modified in the working tree, a file *dir/file2* added to the working tree, but also a file *dir/file3* removed from the working tree). Note that older versions of Git used to ignore removed files; use *--no-all* option if you want to add modified or new files but ignore removed ones.

For more details about the *<pathspec>* syntax, see the *pathspec* entry in Section G.4.19, “gitglossary(7)”.

-n , --dry-run

Don't actually add the file(s), just show if they exist and/or will be ignored.

-v , --verbose

Be verbose.

-f , --force

Allow adding otherwise ignored files.

--sparse

Allow updating index entries outside of the sparse-checkout cone. Normally, *git add* refuses to update index entries whose paths do not fit within the sparse-checkout cone, since those files might be removed from the working tree without warning. See Section G.3.135, “git-sparse-checkout(1)” for more details.

-i , --interactive

Add modified contents in the working tree interactively to the index. Optional path arguments may be supplied to limit operation to a subset of the working tree. See Interactive mode for details.

-p , --patch

Interactively choose hunks of patch between the index and the work tree and add them to the index. This gives the user a chance to review the difference before adding modified contents to the index.

This effectively runs *add --interactive*, but bypasses the initial command menu and directly jumps to the *patch* subcommand. See Interactive mode for details.

-e , --edit

Open the diff vs. the index in an editor and let the user edit it. After the editor was closed, adjust the hunk headers and apply the patch to the index.

The intent of this option is to pick and choose lines of the patch to apply, or even to modify the contents of lines to be staged. This can be quicker and more flexible than using the interactive hunk selector. However, it is easy to confuse oneself and create a patch that does not apply to the index. See EDITING PATCHES below.

-u , --update

Update the index just where it already has an entry matching <*pathspec*>. This removes as well as modifies index entries to match the working tree, but adds no new files.

If no <*pathspec*> is given when *-u* option is used, all tracked files in the entire working tree are updated (old versions of Git used to limit the update to the current directory and its subdirectories).

-A , --all , --no-ignore-removal

Update the index not only where the working tree has a file matching <*pathspec*> but also where the index already has an entry. This adds, modifies, and removes index entries to match the working tree.

If no <*pathspec*> is given when *-A* option is used, all files in the entire working tree are updated (old versions of Git used to limit the update to the current directory and its subdirectories).

--no-all , --ignore-removal

Update the index by adding new files that are unknown to the index and files modified in the working tree, but ignore files that have been removed from the working tree. This option is a no-op when no <*pathspec*> is used.

This option is primarily to help users who are used to older versions of Git, whose "git add <*pathspec*>…" was a synonym for "git add --no-all <*pathspec*>…", i.e. ignored removed files.

-N , --intent-to-add

Record only the fact that the path will be added later. An entry for the path is placed in the index with no content. This is useful for, among other things, showing the unstaged content of such files with *git diff* and committing them with *git commit -a*.

--refresh

Don't add the file(s), but only refresh their stat() information in the index.

--ignore-errors

If some files could not be added because of errors indexing them, do not abort the operation, but continue adding the others. The command shall still exit with non-zero status. The configuration variable *add.ignoreErrors* can be set to true to make this the default behaviour.

--ignore-missing

This option can only be used together with --dry-run. By using this option the user can check if any of the given files would be ignored, no matter if they are already present in the work tree or not.

--no-warn-embedded-repo

By default, *git add* will warn when adding an embedded repository to the index without using *git submodule add* to create an entry in *.gitmodules*. This option will suppress the warning (e.g., if you are manually performing operations on submodules).

--renormalize

Apply the "clean" process freshly to all tracked files to forcibly add them again to the index. This is useful after changing *core.autocrlf* configuration or the *text* attribute in order to correct files added with wrong CRLF/LF line endings. This option implies *-u*. Lone CR characters are untouched, thus while a CRLF cleans to LF, a CRCRLF sequence is only partially cleaned to CRLF.

--chmod=(+|-)x

Override the executable bit of the added files. The executable bit is only changed in the index, the files on disk are left unchanged.

--pathspec-from-file=<file>

Pathspec is passed in *<file>* instead of commandline args. If *<file>* is exactly - then standard input is used. Pathspec elements are separated by LF or CR/LF. Pathspec elements can be quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). See also *--pathspec-file-nul* and global *--literal-pathspecs*.

--pathspec-file-nul

Only meaningful with *--pathspec-from-file*. Pathspec elements are separated with NUL character and all other characters are taken literally (including newlines and quotes).

--

This option can be used to separate command-line options from the list of files, (useful when filenames might be mistaken for command-line options).

EXAMPLES

- Adds content from all **.txt* files under *Documentation* directory and its subdirectories:

  ```
  $ git add Documentation/\*.txt
  ```

  Note that the asterisk * is quoted from the shell in this example; this lets the command include the files from subdirectories of *Documentation/* directory.

- Considers adding content from all git-*.sh scripts:

  ```
  $ git add git-*.sh
  ```

  Because this example lets the shell expand the asterisk (i.e. you are listing the files explicitly), it does not consider *subdir/git-foo.sh*.

INTERACTIVE MODE

When the command enters the interactive mode, it shows the output of the *status* subcommand, and then goes into its interactive command loop.

The command loop shows the list of subcommands available, and gives a prompt "What now> ". In general, when the prompt ends with a single >, you can pick only one of the choices given and type return, like this:

```
*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now> 1
```

You also could say *s* or *sta* or *status* above as long as the choice is unique.

The main command loop has 6 subcommands (plus help and quit).

status

This shows the change between HEAD and index (i.e. what will be committed if you say *git commit*), and between index and working tree files (i.e. what you could stage further before *git commit* using *git add*) for each path. A sample output looks like this:

```
              staged     unstaged path
     1:       binary      nothing foo.png
     2:     +403/-35        +1/-1 add-interactive.c
```

It shows that foo.png has differences from HEAD (but that is binary so line count cannot be shown) and there is no difference between indexed copy and the working tree version (if the working tree version were also different, *binary* would have been shown in place of *nothing*). The other file, add-interactive.c, has 403 lines added and 35 lines deleted if you commit what is in the index, but working tree file has further modifications (one addition and one deletion).

update

This shows the status information and issues an "Update>>" prompt. When the prompt ends with double >>, you can make more than one selection, concatenated with whitespace or comma. Also you can say ranges. E.g. "2-5 7,9" to choose 2,3,4,5,7,9 from the list. If the second number in a range is omitted, all remaining patches are taken. E.g. "7-" to choose 7,8,9 from the list. You can say * to choose everything.

What you chose are then highlighted with *, like this:

```
           staged     unstaged path
  1:       binary      nothing foo.png
* 2:     +403/-35        +1/-1 add-interactive.c
```

To remove selection, prefix the input with - like this:

```
Update>> -2
```

After making the selection, answer with an empty line to stage the contents of working tree files for selected paths in the index.

revert

This has a very similar UI to *update*, and the staged information for selected paths are reverted to that of the HEAD version. Reverting new paths makes them untracked.

add untracked

This has a very similar UI to *update* and *revert*, and lets you add untracked paths to the index.

patch

This lets you choose one path out of a *status* like selection. After choosing the path, it presents the diff between the index and the working tree file and asks you if you want to stage the change of each hunk. You can select one of the following options and type return:

```
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
```

```
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
p - print the current hunk
? - print help
```

After deciding the fate for all hunks, if there is any hunk that was chosen, the index is updated with the selected hunks.

You can omit having to type return here, by setting the configuration variable *interactive.singleKey* to *true*.

diff

This lets you review what will be committed (i.e. between HEAD and index).

EDITING PATCHES

Invoking *git add -e* or selecting *e* from the interactive hunk selector will open a patch in your editor; after the editor exits, the result is applied to the index. You are free to make arbitrary changes to the patch, but note that some changes may have confusing results, or even result in a patch that cannot be applied. If you want to abort the operation entirely (i.e., stage nothing new in the index), simply delete all lines of the patch. The list below describes some common things you may see in a patch, and which editing operations make sense on them.

added content

Added content is represented by lines beginning with "+". You can prevent staging any addition lines by deleting them.

removed content

Removed content is represented by lines beginning with "-". You can prevent staging their removal by converting the "-" to a " " (space).

modified content

Modified content is represented by "-" lines (removing the old content) followed by "+" lines (adding the replacement content). You can prevent staging the modification by converting "-" lines to " ", and removing "+" lines. Beware that modifying only half of the pair is likely to introduce confusing changes to the index.

There are also more complex operations that can be performed. But beware that because the patch is applied only to the index and not the working tree, the working tree will appear to "undo" the change in the index. For example, introducing a new line into the index that is in neither the HEAD nor the working tree will stage the new line for commit, but the line will appear to be reverted in the working tree.

Avoid using these constructs, or do so with extreme caution.

removing untouched content

Content which does not differ between the index and working tree may be shown on context lines, beginning with a " " (space). You can stage context lines for removal by converting the space to a "-". The resulting working tree file will appear to re-add the content.

modifying existing content

One can also modify context lines by staging them for removal (by converting " " to "-") and adding a "+" line with the new content. Similarly, one can modify "+" lines for existing additions or modifications. In all cases, the new modification will appear reverted in the working tree.

new content

You may also add new content that does not exist in the patch; simply add new lines, each starting with "+". The addition will appear reverted in the working tree.

There are also several operations which should be avoided entirely, as they will make the patch impossible to apply:

- adding context (" ") or removal ("-") lines
- deleting context or removal lines
- modifying the contents of context or removal lines

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

add.ignoreErrors , add.ignore-errors (deprecated)

Tells *git add* to continue adding files when some files cannot be added due to indexing errors. Equivalent to the *--ignore-errors* option of Section G.3.2, “git-add(1)”. *add.ignore-errors* is deprecated, as it does not follow the usual naming convention for configuration variables.

add.interactive.useBuiltin

Unused configuration variable. Used in Git versions v2.25.0 to v2.36.0 to enable the built-in version of Section G.3.2, “git-add(1)”'s interactive mode, which then became the default in Git versions v2.37.0 to v2.39.0.

SEE ALSO

Section G.3.138, “git-status(1)” Section G.3.123, “git-rm(1)” Section G.3.118, “git-reset(1)” Section G.3.91, “git-mv(1)” Section G.3.28, “git-commit(1)” Section G.3.147, “git-update-index(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.3. git-am(1)

2024-07-02

Revision History

2024-07-02

NAME

git-am - Apply a series of patches from a mailbox

SYNOPSIS

```
git am [--signoff] [--keep] [--[no-]keep-cr] [--[no-]utf8] [--no-verify]
       [--[no-]3way] [--interactive] [--committer-date-is-author-date]
       [--ignore-date] [--ignore-space-change | --ignore-whitespace]
       [--whitespace=<action>] [-C<n>] [-p<n>] [--directory=<dir>]
       [--exclude=<path>] [--include=<path>] [--reject] [-q | --quiet]
       [--[no-]scissors] [-S[<keyid>]] [--patch-format=<format>]
       [--quoted-cr=<action>]
       [--empty=(stop|drop|keep)]
       [(<mbox> | <Maildir>)…]
git am (--continue | --skip | --abort | --quit | --show-current-patch[=(diff|raw)] | --allow-empty)
```

DESCRIPTION

Splits mail messages in a mailbox into commit log messages, authorship information, and patches, and applies them to the current branch. You could think of it as a reverse operation of Section G.3.54, “git-format-patch(1)” run on a branch with a straight history without merges.

OPTIONS

(<mbox>|<Maildir>)…

The list of mailbox files to read patches from. If you do not supply this argument, the command reads from the standard input. If you supply directories, they will be treated as Maildirs.

-s , --signoff

Add a *Signed-off-by* trailer to the commit message, using the committer identity of yourself. See the signoff option in Section G.3.28, “git-commit(1)” for more information.

-k , --keep

Pass *-k* flag to *git mailinfo* (see Section G.3.78, “git-mailinfo(1)”).

--keep-non-patch

Pass *-b* flag to *git mailinfo* (see Section G.3.78, “git-mailinfo(1)”).

--[no-]keep-cr

With *--keep-cr*, call *git mailsplit* (see Section G.3.79, “git-mailsplit(1)”) with the same option, to prevent it from stripping CR at the end of lines. *am.keepcr* configuration variable can be used to specify the default behaviour. *--no-keep-cr* is useful to override *am.keepcr*.

-c , --scissors

Remove everything in body before a scissors line (see Section G.3.78, “git-mailinfo(1)”). Can be activated by default using the *mailinfo.scissors* configuration variable.

--no-scissors

Ignore scissors lines (see Section G.3.78, “git-mailinfo(1)”).

--quoted-cr=<action>

This flag will be passed down to *git mailinfo* (see Section G.3.78, “git-mailinfo(1)”).

--empty=(drop|keep|stop)

How to handle an e-mail message lacking a patch:

drop

The e-mail message will be skipped.

keep

An empty commit will be created, with the contents of the e-mail message as its log.

stop

The command will fail, stopping in the middle of the current *am* session. This is the default behavior.

-m , --message-id

Pass the *-m* flag to *git mailinfo* (see Section G.3.78, “git-mailinfo(1)”), so that the Message-ID header is added to the commit message. The *am.messageid* configuration variable can be used to specify the default behaviour.

--no-message-id

Do not add the Message-ID header to the commit message. *no-message-id* is useful to override *am.messageid*.

-q , --quiet

Be quiet. Only print error messages.

-u , --utf8

Pass *-u* flag to *git mailinfo* (see Section G.3.78, “git-mailinfo(1)”). The proposed commit log message taken from the e-mail is re-coded into UTF-8 encoding (configuration variable *i18n.commitEncoding* can be used to specify the project's preferred encoding if it is not UTF-8).

This was optional in prior versions of git, but now it is the default. You can use *--no-utf8* to override this.

--no-utf8

Pass *-n* flag to *git mailinfo* (see Section G.3.78, “git-mailinfo(1)”).

-3 , --3way , --no-3way

When the patch does not apply cleanly, fall back on 3-way merge if the patch records the identity of blobs it is supposed to apply to and we have those blobs available locally. *--no-3way* can be used to override am.threeWay configuration variable. For more information, see am.threeWay in Section G.3.29, “git-config(1)”.

--rerere-autoupdate , --no-rerere-autoupdate

After the rerere mechanism reuses a recorded resolution on the current conflict to update the files in the working tree, allow it to also update the index with the result of resolution. *--no-rerere-autoupdate* is a good way to double-check what *rerere* did and catch potential mismerges, before committing the result to the index with a separate *git add*.

--ignore-space-change , --ignore-whitespace , --whitespace=<action> , -C<n> , -p<n> , --directory=<dir> , --exclude=<path> , --include=<path> , --reject

These flags are passed to the *git apply* (see Section G.3.5, “git-apply(1)”) program that applies the patch.

Valid <action> for the *--whitespace* option are: *nowarn*, *warn*, *fix*, *error*, and *error-all*.

--patch-format

By default the command will try to detect the patch format automatically. This option allows the user to bypass the automatic detection and specify the patch format that the patch(es) should be interpreted as. Valid formats are mbox, mboxrd, stgit, stgit-series, and hg.

-i , --interactive

Run interactively.

-n , --no-verify

By default, the pre-applypatch and applypatch-msg hooks are run. When any of *--no-verify* or *-n* is given, these are bypassed. See also Section G.4.7, “githooks(5)”.

--committer-date-is-author-date

By default the command records the date from the e-mail message as the commit author date, and uses the time of commit creation as the committer date. This allows the user to lie about the committer date by using the same value as the author date.

--ignore-date

By default the command records the date from the e-mail message as the commit author date, and uses the time of commit creation as the committer date. This allows the user to lie about the author date by using the same value as the committer date.

--skip

Skip the current patch. This is only meaningful when restarting an aborted patch.

-S[<keyid>] , --gpg-sign[=<keyid>] , --no-gpg-sign

GPG-sign commits. The *keyid* argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space. *--no-gpg-sign* is useful to countermand both *commit.gpgSign* configuration variable, and earlier *--gpg-sign*.

--continue , -r , --resolved

After a patch failure (e.g. attempting to apply conflicting patch), the user has applied it by hand and the index file stores the result of the application. Make a commit using the authorship and commit log extracted from the e-mail message and the current index file, and continue.

--resolvemsg=<msg>

When a patch failure occurs, <msg> will be printed to the screen before exiting. This overrides the standard message informing you to use *--continue* or *--skip* to handle the failure. This is solely for internal use between *git rebase* and *git am*.

--abort

Restore the original branch and abort the patching operation. Revert the contents of files involved in the am operation to their pre-am state.

--quit

Abort the patching operation but keep HEAD and the index untouched.

--show-current-patch[=(diff|raw)]

Show the message at which *git am* has stopped due to conflicts. If *raw* is specified, show the raw contents of the e-mail message; if *diff*, show the diff portion only. Defaults to *raw*.

--allow-empty

After a patch failure on an input e-mail message lacking a patch, create an empty commit with the contents of the e-mail message as its log message.

DISCUSSION

The commit author name is taken from the "From: " line of the message, and commit author date is taken from the "Date: " line of the message. The "Subject: " line is used as the title of the commit, after stripping common prefix "[PATCH <anything>]". The "Subject: " line is supposed to concisely describe what the commit is about in one line of text.

"From: ", "Date: ", and "Subject: " lines starting the body override the respective commit author name and title values taken from the headers.

The commit message is formed by the title taken from the "Subject: ", a blank line and the body of the message up to where the patch begins. Excess whitespace at the end of each line is automatically stripped.

The patch is expected to be inline, directly following the message. Any line that is of the form:

- three-dashes and end-of-line, or
- a line that begins with "diff -", or
- a line that begins with "Index: "

is taken as the beginning of a patch, and the commit log message is terminated before the first occurrence of such a line.

When initially invoking *git am*, you give it the names of the mailboxes to process. Upon seeing the first patch that does not apply, it aborts in the middle. You can recover from this in one of two ways:

1. skip the current patch by re-running the command with the *--skip* option.
2. hand resolve the conflict in the working directory, and update the index file to bring it into a state that the patch should have produced. Then run the command with the *--continue* option.

The command refuses to process new mailboxes until the current operation is finished, so if you decide to start over from scratch, run *git am --abort* before running the command with mailbox names.

Before any patches are applied, ORIG_HEAD is set to the tip of the current branch. This is useful if you have problems with multiple commits, like running *git am* on the wrong branch or an error in the commits that is more easily fixed by changing the mailbox (e.g. errors in the "From:" lines).

HOOKS

This command can run *applypatch-msg*, *pre-applypatch*, and *post-applypatch* hooks. See Section G.4.7, “githooks(5)” for more information.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

am.keepcr

If true, git-am will call git-mailsplit for patches in mbox format with parameter *--keep-cr*. In this case git-mailsplit will not remove *\r* from lines ending with *\r\n*. Can be overridden by giving *--no-keep-cr* from the command line. See Section G.3.3, “git-am(1)”, Section G.3.79, “git-mailsplit(1)”.

am.threeWay

By default, *git am* will fail if the patch does not apply cleanly. When set to true, this setting tells *git am* to fall back on 3-way merge if the patch records the identity of blobs it is supposed to apply to and we have those blobs available locally (equivalent to giving the *--3way* option from the command line). Defaults to *false*. See Section G.3.3, “git-am(1)”.

SEE ALSO

Section G.3.5, “git-apply(1)”, Section G.3.54, “git-format-patch(1)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.4. git-annotate(1)

2024-07-02

Revision History

2024-07-02

NAME

git-annotate - Annotate file lines with commit information

SYNOPSIS

git annotate [<options>] [<rev-opts>] [<rev>] [--] <file>

DESCRIPTION

Annotates each line in the given file with information from the commit which introduced the line. Optionally annotates from a given revision.

The only difference between this command and Section G.3.9, “git-blame(1)” is that they use slightly different output formats, and this command exists only for backward compatibility to support existing scripts, and provide a more familiar command name for people coming from other SCM systems.

OPTIONS

-b

Show blank SHA-1 for boundary commits. This can also be controlled via the *blame.blankBoundary* config option.

--root

Do not treat root commits as boundaries. This can also be controlled via the *blame.showRoot* config option.

--show-stats

Include additional statistics at the end of blame output.

-L <start>,<end> , -L :<funcname>

Annotate only the line range given by *<start>*,*<end>*, or by the function name regex *<funcname>*. May be specified multiple times. Overlapping ranges are allowed.

<start> and *<end>* are optional. *-L <start>* or *-L <start>*, spans from *<start>* to end of file. *-L ,<end>* spans from start of file to *<end>*.

<start> and *<end>* can take one of these forms:

- number

  If *<start>* or *<end>* is a number, it specifies an absolute line number (lines count from 1).

- */regex/*

  This form will use the first line matching the given POSIX regex. If *<start>* is a regex, it will search from the end of the previous *-L* range, if any, otherwise from the start of file. If *<start>* is *^/regex/*, it will search from the start of file. If *<end>* is a regex, it will search starting at the line given by *<start>*.

- +offset or -offset

  This is only valid for *<end>* and will specify a number of lines before or after the line given by *<start>*.

If *:<funcname>* is given in place of *<start>* and *<end>*, it is a regular expression that denotes the range from the first funcname line that matches *<funcname>*, up to the next funcname line. *:<funcname>* searches from the end of the previous *-L* range, if any, otherwise from the start of file. *^:<funcname>* searches from the start of file. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

-l

Show long rev (Default: off).

-t

Show raw timestamp (Default: off).

-S <revs-file>

Use revisions from revs-file instead of calling Section G.3.120, “git-rev-list(1)”.

--reverse <rev>..<rev>

Walk history forward instead of backward. Instead of showing the revision in which a line appeared, this shows the last revision in which a line has existed. This requires a range of revision like START..END where the path to blame exists in START. *git blame --reverse START* is taken as *git blame --reverse START..HEAD* for convenience.

--first-parent

Follow only the first parent commit upon seeing a merge commit. This option can be used to determine when a line was introduced to a particular integration branch, rather than when it was introduced to the history overall.

-p , --porcelain

Show in a format designed for machine consumption.

--line-porcelain

Show the porcelain format, but output commit information for each line, not just the first time a commit is referenced. Implies --porcelain.

--incremental

Show the result incrementally in a format designed for machine consumption.

--encoding=<encoding>

Specifies the encoding used to output author names and commit summaries. Setting it to *none* makes blame output unconverted data. For more information see the discussion about encoding in the Section G.3.74, “git-log(1)” manual page.

--contents <file>

Annotate using the contents from the named file, starting from <rev> if it is specified, and HEAD otherwise. You may specify - to make the command read from the standard input for the file contents.

--date <format>

Specifies the format used to output dates. If --date is not provided, the value of the blame.date config variable is used. If the blame.date config variable is also not set, the iso format is used. For supported values, see the discussion of the --date option at Section G.3.74, “git-log(1)”.

--[no-]progress

Progress status is reported on the standard error stream by default when it is attached to a terminal. This flag enables progress reporting even if not attached to a terminal. Can't use *--progress* together with *--porcelain* or *--incremental*.

-M[<num>]

Detect moved or copied lines within a file. When a commit moves or copies a block of lines (e.g. the original file has A and then B, and the commit changes it to B and then A), the traditional *blame* algorithm notices only half of the movement and typically blames the lines that were moved up (i.e. B) to the parent and assigns blame to the lines that were moved down (i.e. A) to the child commit. With this option, both groups of lines are blamed on the parent by running extra passes of inspection.

<num> is optional but it is the lower bound on the number of alphanumeric characters that Git must detect as moving/copying within a file for it to associate those lines with the parent commit. The default value is 20.

-C[<num>]

In addition to *-M*, detect lines moved or copied from other files that were modified in the same commit. This is useful when you reorganize your program and move code around across files. When this option is given twice, the command additionally looks for copies from other files in the commit that creates the file. When this option is given three times, the command additionally looks for copies from other files in any commit.

<num> is optional but it is the lower bound on the number of alphanumeric characters that Git must detect as moving/copying between files for it to associate those lines with the parent commit. And the default value is 40. If there are more than one *-C* options given, the <num> argument of the last *-C* will take effect.

--ignore-rev <rev>

Ignore changes made by the revision when assigning blame, as if the change never happened. Lines that were changed or added by an ignored commit will be blamed on the previous commit that changed that line or nearby lines. This option may be specified multiple times to ignore more than one revision. If the *blame.markIgnoredLines* config option is set, then lines that were changed by an ignored commit and attributed to another commit will be marked with a *?* in the blame output. If the *blame.markUnblamableLines* config option is set, then those lines touched by an ignored commit that we could not attribute to another revision are marked with a *.

--ignore-revs-file <file>

Ignore revisions listed in *file*, which must be in the same format as an *fsck.skipList*. This option may be repeated, and these files will be processed after any files specified with the *blame.ignoreRevsFile* config option. An empty file name, *""*, will clear the list of revs from previously processed files.

--color-lines

Color line annotations in the default format differently if they come from the same commit as the preceding line. This makes it easier to distinguish code blocks introduced by different commits. The color defaults to cyan and can be adjusted using the *color.blame.repeatedLines* config option.

--color-by-age

Color line annotations depending on the age of the line in the default format. The *color.blame.highlightRecent* config option controls what color is used for each range of age.

-h

Show help message.

SEE ALSO

Section G.3.9, “git-blame(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.5. git-apply(1)

2024-07-02

Revision History

2024-07-02

NAME

git-apply - Apply a patch to files and/or to the index

SYNOPSIS

git apply [--stat] [--numstat] [--summary] [--check] [--index | --intent-to-add] [--3way]

[--apply] [--no-add] [--build-fake-ancestor=<file>] [-R | --reverse]
[--allow-binary-replacement | --binary] [--reject] [-z]
[-p<n>] [-C<n>] [--inaccurate-eof] [--recount] [--cached]
[--ignore-space-change | --ignore-whitespace]
[--whitespace=(nowarn|warn|fix|error|error-all)]
[--exclude=<path>] [--include=<path>] [--directory=<root>]
[--verbose | --quiet] [--unsafe-paths] [--allow-empty] [<patch>…]

DESCRIPTION

Reads the supplied diff output (i.e. "a patch") and applies it to files. When running from a subdirectory in a repository, patched paths outside the directory are ignored. With the *--index* option, the patch is also applied to the index, and with the *--cached* option, the patch is only applied to the index. Without these options, the command applies the patch only to files, and does not require them to be in a Git repository.

This command applies the patch but does not create a commit. Use Section G.3.3, “git-am(1)” to create commits from patches generated by Section G.3.54, “git-format-patch(1)” and/or received by email.

OPTIONS

<patch>…

The files to read the patch from. - can be used to read from the standard input.

--stat

Instead of applying the patch, output diffstat for the input. Turns off "apply".

--numstat

Similar to *--stat*, but shows the number of added and deleted lines in decimal notation and the pathname without abbreviation, to make it more machine friendly. For binary files, outputs two - instead of saying *0 0*. Turns off "apply".

--summary

Instead of applying the patch, output a condensed summary of information obtained from git diff extended headers, such as creations, renames, and mode changes. Turns off "apply".

--check

Instead of applying the patch, see if the patch is applicable to the current working tree and/or the index file and detects errors. Turns off "apply".

--index

Apply the patch to both the index and the working tree (or merely check that it would apply cleanly to both if *--check* is in effect). Note that *--index* expects index entries and working tree copies for relevant paths to be identical (their contents and metadata such as file mode must match), and will raise an error if they are not, even if the patch would apply cleanly to both the index and the working tree in isolation.

--cached

Apply the patch to just the index, without touching the working tree. If *--check* is in effect, merely check that it would apply cleanly to the index entry.

--intent-to-add

When applying the patch only to the working tree, mark new files to be added to the index later (see *--intent-to-add* option in Section G.3.2, “git-add(1)”). This option is ignored unless running in a Git repository and *--index* is not specified. Note that *--index* could be implied by other options such as *--cached* or *--3way*.

-3 , --3way

Attempt 3-way merge if the patch records the identity of blobs it is supposed to apply to and we have those blobs available locally, possibly leaving the conflict markers in the files in the working tree for the user to resolve. This option implies the *--index* option unless the *--cached* option is used, and is incompatible with the *--reject* option. When used with the *--cached* option, any conflicts are left at higher stages in the cache.

--build-fake-ancestor=<file>

Newer *git diff* output has embedded *index information* for each blob to help identify the original version that the patch applies to. When this flag is given, and if the original versions of the blobs are available locally, builds a temporary index containing those blobs.

When a pure mode change is encountered (which has no index information), the information is read from the current index instead.

-R , --reverse

Apply the patch in reverse.

--reject

For atomicity, *git apply* by default fails the whole patch and does not touch the working tree when some of the hunks do not apply. This option makes it apply the parts of the patch that are applicable, and leave the rejected hunks in corresponding *.rej files.

-z

When *--numstat* has been given, do not munge pathnames, but use a NUL-terminated machine-readable format.

Without this option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

-p<n>

Remove <n> leading path components (separated by slashes) from traditional diff paths. E.g., with *-p2*, a patch against *a/dir/file* will be applied directly to *file*. The default is 1.

-C<n>

Ensure at least <n> lines of surrounding context match before and after each change. When fewer lines of surrounding context exist they all must match. By default no context is ever ignored.

--unidiff-zero

By default, *git apply* expects that the patch being applied is a unified diff with at least one line of context. This provides good safety measures, but breaks down when applying a diff generated with *--unified=0*. To bypass these checks use *--unidiff-zero*.

Note, for the reasons stated above, the usage of context-free patches is discouraged.

--apply

If you use any of the options marked "Turns off *apply*" above, *git apply* reads and outputs the requested information without actually applying the patch. Give this flag after those flags to also apply the patch.

--no-add

When applying a patch, ignore additions made by the patch. This can be used to extract the common part between two files by first running *diff* on them and applying the result with this option, which would apply the deletion part but not the addition part.

--allow-binary-replacement , --binary

Historically we did not allow binary patch application without an explicit permission from the user, and this flag was the way to do so. Currently, we always allow binary patch application, so this is a no-op.

--exclude=<path-pattern>

Don't apply changes to files matching the given path pattern. This can be useful when importing patchsets, where you want to exclude certain files or directories.

--include=<path-pattern>

Apply changes to files matching the given path pattern. This can be useful when importing patchsets, where you want to include certain files or directories.

When *--exclude* and *--include* patterns are used, they are examined in the order they appear on the command line, and the first match determines if a patch to each path is used. A patch to a path that does not match any include/exclude pattern is used by default if there is no include pattern on the command line, and ignored if there is any include pattern.

--ignore-space-change , --ignore-whitespace

When applying a patch, ignore changes in whitespace in context lines if necessary. Context lines will preserve their whitespace, and they will not undergo whitespace fixing regardless of the value of the *--whitespace* option. New lines will still be fixed, though.

--whitespace=<action>

When applying a patch, detect a new or modified line that has whitespace errors. What are considered white-space errors is controlled by *core.whitespace* configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors.

By default, the command outputs warning messages but applies the patch. When *git-apply* is used for statistics and not applying a patch, it defaults to *nowarn*.

You can use different <*action*> values to control this behavior:

- *nowarn* turns off the trailing whitespace warning.
- *warn* outputs warnings for a few such errors, but applies the patch as-is (default).
- *fix* outputs warnings for a few such errors, and applies the patch after fixing them (*strip* is a synonym -- the tool used to consider only trailing whitespace characters as errors, and the fix involved *stripping* them, but modern Gits do more).
- *error* outputs warnings for a few such errors, and refuses to apply the patch.
- *error-all* is similar to *error* but shows all errors.

--inaccurate-eof

Under certain circumstances, some versions of *diff* do not correctly detect a missing new-line at the end of the file. As a result, patches created by such *diff* programs do not record incomplete lines correctly. This option adds support for applying such patches by working around this bug.

-v , --verbose

Report progress to stderr. By default, only a message about the current patch being applied will be printed. This option will cause additional information to be reported.

-q , --quiet

Suppress stderr output. Messages about patch status and progress will not be printed.

--recount

Do not trust the line counts in the hunk headers, but infer them by inspecting the patch (e.g. after editing the patch without adjusting the hunk headers appropriately).

--directory=<root>

Prepend <root> to all filenames. If a "-p" argument was also passed, it is applied before prepending the new root.

For example, a patch that talks about updating *a/git-gui.sh* to *b/git-gui.sh* can be applied to the file in the working tree *modules/git-gui/git-gui.sh* by running *git apply --directory=modules/git-gui*.

--unsafe-paths

By default, a patch that affects outside the working area (either a Git controlled working tree, or the current working directory when "git apply" is used as a replacement of GNU patch) is rejected as a mistake (or a mischief).

When *git apply* is used as a "better GNU patch", the user can pass the *--unsafe-paths* option to override this safety check. This option has no effect when *--index* or *--cached* is in use.

--allow-empty

Don't return an error for patches containing no diff. This includes empty patches and patches with commit text only.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

apply.ignoreWhitespace

When set to *change*, tells *git apply* to ignore changes in whitespace, in the same way as the *--ignore-space-change* option. When set to one of: no, none, never, false, it tells *git apply* to respect all whitespace differences. See Section G.3.5, “git-apply(1)”.

apply.whitespace

Tells *git apply* how to handle whitespace, in the same way as the *--whitespace* option. See Section G.3.5, “git-apply(1)”.

SUBMODULES

If the patch contains any changes to submodules then *git apply* treats these changes as follows.

If *--index* is specified (explicitly or implicitly), then the submodule commits must match the index exactly for the patch to apply. If any of the submodules are checked-out, then these check-outs are completely ignored, i.e., they are not required to be up to date or clean and they are not updated.

If *--index* is not specified, then the submodule commits in the patch are ignored and only the absence or presence of the corresponding subdirectory is checked and (if possible) updated.

SEE ALSO

Section G.3.3, “git-am(1)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.6. git-archimport(1)

2024-07-02

Revision History

2024-07-02

NAME

git-archimport - Import a GNU Arch repository into Git

SYNOPSIS

git archimport [-h] [-v] [-o] [-a] [-f] [-T] [-D <depth>] [-t <tempdir>]
<archive>/<branch>[:<git-branch>]…

DESCRIPTION

Imports a project from one or more GNU Arch repositories. It will follow branches and repositories within the namespaces defined by the <archive>/<branch> parameters supplied. If it cannot find the remote branch a merge comes from it will just import it as a regular commit. If it can find it, it will mark it as a merge whenever possible (see discussion below).

The script expects you to provide the key roots where it can start the import from an *initial import* or *tag* type of Arch commit. It will follow and import new branches within the provided roots.

It expects to be dealing with one project only. If it sees branches that have different roots, it will refuse to run. In that case, edit your <archive>/<branch> parameters to define clearly the scope of the import.

git archimport uses *tla* extensively in the background to access the Arch repository. Make sure you have a recent version of *tla* available in the path. *tla* must know about the repositories you pass to *git archimport*.

For the initial import, *git archimport* expects to find itself in an empty directory. To follow the development of a project that uses Arch, rerun *git archimport* with the same parameters as the initial import to perform incremental imports.

While *git archimport* will try to create sensible branch names for the archives that it imports, it is also possible to specify Git branch names manually. To do so, write a Git branch name after each <archive>/<branch> parameter, separated by a colon. This way, you can shorten the Arch branch names and convert Arch jargon to Git jargon, for example mapping a "PROJECT--devo--VERSION" branch to "master".

Associating multiple Arch branches to one Git branch is possible; the result will make the most sense only if no commits are made to the first branch, after the second branch is created. Still, this is useful to convert Arch repositories that had been rotated periodically.

MERGES

Patch merge data from Arch is used to mark merges in Git as well. Git does not care much about tracking patches, and only considers a merge when a branch incorporates all the commits since the point they forked. The end result is that Git will have a good idea of how far branches have diverged. So the import process does lose some patch-trading metadata.

Fortunately, when you try and merge branches imported from Arch, Git will find a good merge base, and it has a good chance of identifying patches that have been traded out-of-sequence between the branches.

OPTIONS

-h

Display usage.

-v

Verbose output.

-T

Many tags. Will create a tag for every commit, reflecting the commit name in the Arch repository.

-f

Use the fast patchset import strategy. This can be significantly faster for large trees, but cannot handle directory renames or permissions changes. The default strategy is slow and safe.

-o

Use this for compatibility with old-style branch names used by earlier versions of *git archimport*. Old-style branch names were category--branch, whereas new-style branch names are archive,category--branch--version. In both cases, names given on the command-line will override the automatically-generated ones.

-D <depth>

Follow merge ancestry and attempt to import trees that have been merged from. Specify a depth greater than 1 if patch logs have been pruned.

-a

Attempt to auto-register archives at *http://mirrors.sourcecontrol.net* This is particularly useful with the -D option.

-t <tmpdir>

Override the default tempdir.

<archive>/<branch>

<archive>/<branch> identifier in a format that *tla log* understands.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.7. git-archive(1)

2024-07-02
Revision History
2024-07-02

NAME

git-archive - Create an archive of files from a named tree

SYNOPSIS

git archive [--format=<fmt>] [--list] [--prefix=<prefix>/] [<extra>]
[-o <file> | --output=<file>] [--worktree-attributes]
[--remote=<repo> [--exec=<git-upload-archive>]] <tree-ish>
[<path>…]

DESCRIPTION

Creates an archive of the specified format containing the tree structure for the named tree, and writes it out to the standard output. If <prefix> is specified it is prepended to the filenames in the archive.

git archive behaves differently when given a tree ID as opposed to a commit ID or tag ID. When a tree ID is provided, the current time is used as the modification time of each file in the archive. On the other hand, when a commit ID or tag ID is provided, the commit time as recorded in the referenced commit object is used instead. Additionally the commit ID is stored in a global extended pax header if the tar format is used; it can be extracted using *git get-tar-commit-id*. In ZIP files it is stored as a file comment.

OPTIONS

--format=<fmt>

Format of the resulting archive. Possible values are *tar*, *zip*, *tar.gz*, *tgz*, and any format defined using the configuration option *tar.<format>.command*. If *--format* is not given, and the output file is specified, the format is inferred from the filename if possible (e.g. writing to *foo.zip* makes the output to be in the *zip* format). Otherwise the output format is *tar*.

-l , --list

Show all available formats.

-v , --verbose

Report progress to stderr.

--prefix=<prefix>/

Prepend <prefix>/ to paths in the archive. Can be repeated; its rightmost value is used for all tracked files. See below which value gets used by *--add-file* and *--add-virtual-file*.

-o <file> , --output=<file>

Write the archive to <file> instead of stdout.

--add-file=<file>

Add a non-tracked file to the archive. Can be repeated to add multiple files. The path of the file in the archive is built by concatenating the value of the last *--prefix* option (if any) before this *--add-file* and the basename of <file>.

--add-virtual-file=<path>:<content>

Add the specified contents to the archive. Can be repeated to add multiple files. The path of the file in the archive is built by concatenating the value of the last *--prefix* option (if any) before this *--add-virtual-file* and *<path>*.

The *<path>* argument can start and end with a literal double-quote character; the contained file name is interpreted as a C-style string, i.e. the backslash is interpreted as escape character. The path must be quoted if it contains a colon, to avoid the colon from being misinterpreted as the separator between the path and the contents, or if the path begins or ends with a double-quote character.

The file mode is limited to a regular file, and the option may be subject to platform-dependent command-line limits. For non-trivial cases, write an untracked file and use *--add-file* instead.

--worktree-attributes

Look for attributes in .gitattributes files in the working tree as well (see the section called “ATTRIBUTES”).

--mtime=<time>

Set modification time of archive entries. Without this option the committer time is used if *<tree-ish>* is a commit or tag, and the current time if it is a tree.

<extra>

This can be any options that the archiver backend understands. See next section.

--remote=<repo>

Instead of making a tar archive from the local repository, retrieve a tar archive from a remote repository. Note that the remote repository may place restrictions on which sha1 expressions may be allowed in *<tree-ish>*. See Section G.3.150, “git-upload-archive(1)” for details.

--exec=<git-upload-archive>

Used with --remote to specify the path to the *git-upload-archive* on the remote side.

<tree-ish>

The tree or commit to produce an archive for.

<path>

Without an optional path parameter, all files and subdirectories of the current working directory are included in the archive. If one or more paths are specified, only these are included.

BACKEND EXTRA OPTIONS

1. zip

-<digit>

Specify compression level. Larger values allow the command to spend more time to compress to smaller size. Supported values are from *-0* (store-only) to *-9* (best ratio). Default is *-6* if not given.

2. tar

-<number>

Specify compression level. The value will be passed to the compression command configured in *tar.<format>.command*. See manual page of the configured command for the list of supported levels and the default level if this option isn't specified.

CONFIGURATION

tar.umask

This variable can be used to restrict the permission bits of tar archive entries. The default is 0002, which turns off the world write bit. The special value "user" indicates that the archiving user's umask will be used instead. See umask(2) for details. If *--remote* is used then only the configuration of the remote repository takes effect.

tar.<format>.command

This variable specifies a shell command through which the tar output generated by *git archive* should be piped. The command is executed using the shell with the generated tar file on its standard input, and should produce the final output on its standard output. Any compression-level options will be passed to the command (e.g., *-9*).

The *tar.gz* and *tgz* formats are defined automatically and use the magic command *git archive gzip* by default, which invokes an internal implementation of gzip.

tar.<format>.remote

If true, enable the format for use by remote clients via Section G.3.150, “git-upload-archive(1)”. Defaults to false for user-defined formats, but true for the *tar.gz* and *tgz* formats.

ATTRIBUTES

export-ignore

Files and directories with the attribute export-ignore won't be added to archive files. See Section G.4.2, “gitattributes(5)” for details.

export-subst

If the attribute export-subst is set for a file then Git will expand several placeholders when adding this file to an archive. See Section G.4.2, “gitattributes(5)” for details.

Note that attributes are by default taken from the *.gitattributes* files in the tree that is being archived. If you want to tweak the way the output is generated after the fact (e.g. you committed without adding an appropriate export-ignore in its *.gitattributes*), adjust the checked out *.gitattributes* file as necessary and use *--worktree-attributes* option. Alternatively you can keep necessary attributes that should apply while archiving any tree in your *$GIT_DIR/info/attributes* file.

EXAMPLES

git archive --format=tar --prefix=junk/ HEAD | (cd /var/tmp/ && tar xf -)

Create a tar archive that contains the contents of the latest commit on the current branch, and extract it in the */var/tmp/junk* directory.

git archive --format=tar --prefix=git-1.4.0/ v1.4.0 | gzip >git-1.4.0.tar.gz

Create a compressed tarball for v1.4.0 release.

git archive --format=tar.gz --prefix=git-1.4.0/ v1.4.0 >git-1.4.0.tar.gz

Same as above, but using the builtin tar.gz handling.

git archive --prefix=git-1.4.0/ -o git-1.4.0.tar.gz v1.4.0

Same as above, but the format is inferred from the output file.

git archive --format=tar --prefix=git-1.4.0/ v1.4.0^{tree} | gzip >git-1.4.0.tar.gz

Create a compressed tarball for v1.4.0 release, but without a global extended pax header.

git archive --format=zip --prefix=git-docs/ HEAD:Documentation/ > git-1.4.0-docs.zip

Put everything in the current head's Documentation/ directory into *git-1.4.0-docs.zip*, with the prefix *git-docs/*.

git archive -o latest.zip HEAD

Create a Zip archive that contains the contents of the latest commit on the current branch. Note that the output format is inferred by the extension of the output file.

git archive -o latest.tar --prefix=build/ --add-file=configure --prefix= HEAD

Creates a tar archive that contains the contents of the latest commit on the current branch with no prefix and the untracked file *configure* with the prefix *build/*.

git config tar.tar.xz.command "xz -c"

Configure a "tar.xz" format for making LZMA-compressed tarfiles. You can use it specifying *--format=tar.xz*, or by creating an output file like *-o foo.tar.xz*.

SEE ALSO

Section G.4.2, “gitattributes(5)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.8. git-bisect(1)

2024-07-02
Revision History
2024-07-02

NAME

git-bisect - Use binary search to find the commit that introduced a bug

SYNOPSIS

git bisect <subcommand> <options>

DESCRIPTION

The command takes various subcommands, and different options depending on the subcommand:

```
git bisect start [--term-(bad|new)=<term-new> --term-(good|old)=<term-old>]
                 [--no-checkout] [--first-parent] [<bad> [<good>...]] [--]
 [<pathspec>...]
git bisect (bad|new|<term-new>) [<rev>]
git bisect (good|old|<term-old>) [<rev>...]
git bisect terms [--term-(good|old) | --term-(bad|new)]
git bisect skip [(<rev>|<range>)...]
git bisect reset [<commit>]
git bisect (visualize|view)
git bisect replay <logfile>
git bisect log
git bisect run <cmd> [<arg>...]
git bisect help
```

This command uses a binary search algorithm to find which commit in your project's history introduced a bug. You use it by first telling it a "bad" commit that is known to contain the bug, and a "good" commit that is known to be before the bug was introduced. Then *git bisect* picks a commit between those two endpoints and asks you whether the selected commit is "good" or "bad". It continues narrowing down the range until it finds the exact commit that introduced the change.

In fact, *git bisect* can be used to find the commit that changed **any** property of your project; e.g., the commit that fixed a bug, or the commit that caused a benchmark's performance to improve. To support this more general usage, the terms "old" and "new" can be used in place of "good" and "bad", or you can choose your own terms. See section "Alternate terms" below for more information.

1. Basic bisect commands: start, bad, good

As an example, suppose you are trying to find the commit that broke a feature that was known to work in version *v2.6.13-rc2* of your project. You start a bisect session as follows:

```
$ git bisect start
$ git bisect bad                 # Current version is bad
$ git bisect good v2.6.13-rc2    # v2.6.13-rc2 is known to be good
```

Once you have specified at least one bad and one good commit, *git bisect* selects a commit in the middle of that range of history, checks it out, and outputs something similar to the following:

```
Bisecting: 675 revisions left to test after this (roughly 10 steps)
```

You should now compile the checked-out version and test it. If that version works correctly, type

```
$ git bisect good
```

If that version is broken, type

```
$ git bisect bad
```

Then *git bisect* will respond with something like

```
Bisecting: 337 revisions left to test after this (roughly 9 steps)
```

Keep repeating the process: compile the tree, test it, and depending on whether it is good or bad run *git bisect good* or *git bisect bad* to ask for the next commit that needs testing.

Eventually there will be no more revisions left to inspect, and the command will print out a description of the first bad commit. The reference *refs/bisect/bad* will be left pointing at that commit.

2. Bisect reset

After a bisect session, to clean up the bisection state and return to the original HEAD, issue the following command:

```
$ git bisect reset
```

By default, this will return your tree to the commit that was checked out before *git bisect start*. (A new *git bisect start* will also do that, as it cleans up the old bisection state.)

With an optional argument, you can return to a different commit instead:

```
$ git bisect reset <commit>
```

For example, *git bisect reset bisect/bad* will check out the first bad revision, while *git bisect reset HEAD* will leave you on the current bisection commit and avoid switching commits at all.

3. Alternate terms

Sometimes you are not looking for the commit that introduced a breakage, but rather for a commit that caused a change between some other "old" state and "new" state. For example, you might be looking for the commit that introduced a particular fix. Or you might be looking for the first commit in which the source-code filenames were finally all converted to your company's naming standard. Or whatever.

In such cases it can be very confusing to use the terms "good" and "bad" to refer to "the state before the change" and "the state after the change". So instead, you can use the terms "old" and "new", respectively, in place of "good" and "bad". (But note that you cannot mix "good" and "bad" with "old" and "new" in a single session.)

In this more general usage, you provide *git bisect* with a "new" commit that has some property and an "old" commit that doesn't have that property. Each time *git bisect* checks out a commit, you test if that commit has the property. If it does, mark the commit as "new"; otherwise, mark it as "old". When the bisection is done, *git bisect* will report which commit introduced the property.

To use "old" and "new" instead of "good" and bad, you must run *git bisect start* without commits as argument and then run the following commands to add the commits:

```
git bisect old [<rev>]
```

to indicate that a commit was before the sought change, or

```
git bisect new [<rev>...]
```

to indicate that it was after.

To get a reminder of the currently used terms, use

```
git bisect terms
```

You can get just the old term with *git bisect terms --term-old* or *git bisect terms --term-good*; *git bisect terms --term-new* and *git bisect terms --term-bad* can be used to learn how to call the commits more recent than the sought change.

If you would like to use your own terms instead of "bad"/"good" or "new"/"old", you can choose any names you like (except existing bisect subcommands like *reset*, *start*, …) by starting the bisection using

```
git bisect start --term-old <term-old> --term-new <term-new>
```

For example, if you are looking for a commit that introduced a performance regression, you might use

```
git bisect start --term-old fast --term-new slow
```

Or if you are looking for the commit that fixed a bug, you might use

```
git bisect start --term-new fixed --term-old broken
```

Then, use *git bisect <term-old>* and *git bisect <term-new>* instead of *git bisect good* and *git bisect bad* to mark commits.

4. Bisect visualize/view

To see the currently remaining suspects in *gitk*, issue the following command during the bisection process (the subcommand *view* can be used as an alternative to *visualize*):

```
$ git bisect visualize
```

Git detects a graphical environment through various environment variables: *DISPLAY*, which is set in X Window System environments on Unix systems. *SESSIONNAME*, which is set under Cygwin in interactive desktop sessions. *MSYSTEM*, which is set under Msys2 and Git for Windows. *SECURITYSESSIONID*, which may be set on macOS in interactive desktop sessions.

If none of these environment variables is set, *git log* is used instead. You can also give command-line options such as *-p* and *--stat*.

```
$ git bisect visualize --stat
```

5. Bisect log and bisect replay

After having marked revisions as good or bad, issue the following command to show what has been done so far:

```
$ git bisect log
```

If you discover that you made a mistake in specifying the status of a revision, you can save the output of this command to a file, edit it to remove the incorrect entries, and then issue the following commands to return to a corrected state:

```
$ git bisect reset
$ git bisect replay that-file
```

6. Avoiding testing a commit

If, in the middle of a bisect session, you know that the suggested revision is not a good one to test (e.g. it fails to build and you know that the failure does not have anything to do with the bug you are chasing), you can manually select a nearby commit and test that one instead.

For example:

```
$ git bisect good/bad                   # previous round was good or bad.
Bisecting: 337 revisions left to test after this (roughly 9 steps)
$ git bisect visualize                  # oops, that is uninteresting.
$ git reset --hard HEAD~3               # try 3 revisions before what
                                        # was suggested
```

Then compile and test the chosen revision, and afterwards mark the revision as good or bad in the usual manner.

7. Bisect skip

Instead of choosing a nearby commit by yourself, you can ask Git to do it for you by issuing the command:

```
$ git bisect skip                 # Current version cannot be tested
```

However, if you skip a commit adjacent to the one you are looking for, Git will be unable to tell exactly which of those commits was the first bad one.

You can also skip a range of commits, instead of just one commit, using range notation. For example:

```
$ git bisect skip v2.5..v2.6
```

This tells the bisect process that no commit after *v2.5*, up to and including *v2.6*, should be tested.

Note that if you also want to skip the first commit of the range you would issue the command:

```
$ git bisect skip v2.5 v2.5..v2.6
```

This tells the bisect process that the commits between *v2.5* and *v2.6* (inclusive) should be skipped.

8. Cutting down bisection by giving more parameters to bisect start

You can further cut down the number of trials, if you know what part of the tree is involved in the problem you are tracking down, by specifying pathspec parameters when issuing the *bisect start* command:

```
$ git bisect start -- arch/i386 include/asm-i386
```

If you know beforehand more than one good commit, you can narrow the bisect space down by specifying all of the good commits immediately after the bad commit when issuing the *bisect start* command:

```
$ git bisect start v2.6.20-rc6 v2.6.20-rc4 v2.6.20-rc1 --
                   # v2.6.20-rc6 is bad
                   # v2.6.20-rc4 and v2.6.20-rc1 are good
```

9. Bisect run

If you have a script that can tell if the current source code is good or bad, you can bisect by issuing the command:

```
$ git bisect run my_script arguments
```

Note that the script (*my_script* in the above example) should exit with code 0 if the current source code is good/old, and exit with a code between 1 and 127 (inclusive), except 125, if the current source code is bad/new.

Any other exit code will abort the bisect process. It should be noted that a program that terminates via *exit(-1)* leaves $? = 255, (see the exit(3) manual page), as the value is chopped with *& 0377*.

The special exit code 125 should be used when the current source code cannot be tested. If the script exits with this code, the current revision will be skipped (see *git bisect skip* above). 125 was chosen as the highest sensible value to use for this purpose, because 126 and 127 are used by POSIX shells to signal specific error status (127

is for command not found, 126 is for command found but not executable--these details do not matter, as they are normal errors in the script, as far as *bisect run* is concerned).

You may often find that during a bisect session you want to have temporary modifications (e.g. s/#define DEBUG 0/#define DEBUG 1/ in a header file, or "revision that does not have this commit needs this patch applied to work around another problem this bisection is not interested in") applied to the revision being tested.

To cope with such a situation, after the inner *git bisect* finds the next revision to test, the script can apply the patch before compiling, run the real test, and afterwards decide if the revision (possibly with the needed patch) passed the test and then rewind the tree to the pristine state. Finally the script should exit with the status of the real test to let the *git bisect run* command loop determine the eventual outcome of the bisect session.

OPTIONS

--no-checkout

Do not checkout the new working tree at each iteration of the bisection process. Instead just update the reference named *BISECT_HEAD* to make it point to the commit that should be tested.

This option may be useful when the test you would perform in each step does not require a checked out tree.

If the repository is bare, *--no-checkout* is assumed.

--first-parent

Follow only the first parent commit upon seeing a merge commit.

In detecting regressions introduced through the merging of a branch, the merge commit will be identified as introduction of the bug and its ancestors will be ignored.

This option is particularly useful in avoiding false positives when a merged branch contained broken or non-buildable commits, but the merge itself was OK.

EXAMPLES

- Automatically bisect a broken build between v1.2 and HEAD:

```
$ git bisect start HEAD v1.2 --      # HEAD is bad, v1.2 is good
$ git bisect run make                # "make" builds the app
$ git bisect reset                   # quit the bisect session
```

- Automatically bisect a test failure between origin and HEAD:

```
$ git bisect start HEAD origin --    # HEAD is bad, origin is good
$ git bisect run make test           # "make test" builds and tests
$ git bisect reset                   # quit the bisect session
```

- Automatically bisect a broken test case:

```
$ cat ~/test.sh
#!/bin/sh
make || exit 125                     # this skips broken builds
~/check_test_case.sh                 # does the test case pass?
$ git bisect start HEAD HEAD~10 --   # culprit is among the last 10
$ git bisect run ~/test.sh
$ git bisect reset                   # quit the bisect session
```

  Here we use a *test.sh* custom script. In this script, if *make* fails, we skip the current commit. *check_test_case.sh* should *exit 0* if the test case passes, and *exit 1* otherwise.

  It is safer if both *test.sh* and *check_test_case.sh* are outside the repository to prevent interactions between the bisect, make and test processes and the scripts.

- Automatically bisect with temporary modifications (hot-fix):

```
$ cat ~/test.sh
#!/bin/sh

# tweak the working tree by merging the hot-fix branch
# and then attempt a build
if      git merge --no-commit --no-ff hot-fix &&
        make
then
        # run project specific test and report its status
        ~/check_test_case.sh
        status=$?
else
        # tell the caller this is untestable
        status=125
fi

# undo the tweak to allow clean flipping to the next commit
git reset --hard

# return control
exit $status
```

  This applies modifications from a hot-fix branch before each test run, e.g. in case your build or test environment changed so that older revisions may need a fix which newer ones have already. (Make sure the hot-fix branch is based off a commit which is contained in all revisions which you are bisecting, so that the merge does not pull in too much, or use *git cherry-pick* instead of *git merge*.)

- Automatically bisect a broken test case:

```
$ git bisect start HEAD HEAD~10 --   # culprit is among the last 10
$ git bisect run sh -c "make || exit 125; ~/check_test_case.sh"
$ git bisect reset                   # quit the bisect session
```

  This shows that you can do without a run script if you write the test on a single line.

- Locate a good region of the object graph in a damaged repository

```
$ git bisect start HEAD <known-good-commit> [ <boundary-commit> ... ] --
no-checkout
$ git bisect run sh -c '
        GOOD=$(git for-each-ref "--format=%(objectname)" refs/bisect/
good-*) &&
        git rev-list --objects BISECT_HEAD --not $GOOD >tmp.$$ &&
        git pack-objects --stdout >/dev/null <tmp.$$
        rc=$?
        rm -f tmp.$$
        test $rc = 0'

$ git bisect reset                   # quit the bisect session
```

  In this case, when *git bisect run* finishes, bisect/bad will refer to a commit that has at least one parent whose reachable graph is fully traversable in the sense required by *git pack objects*.

- Look for a fix instead of a regression in the code

```
$ git bisect start
$ git bisect new HEAD    # current commit is marked as new
$ git bisect old HEAD~10 # the tenth commit from now is marked as old
```

or:

```
$ git bisect start --term-old broken --term-new fixed
$ git bisect fixed
$ git bisect broken HEAD~10
```

1. Getting help

Use *git bisect* to get a short usage description, and *git bisect help* or *git bisect -h* to get a long usage description.

SEE ALSO

Fighting regressions with git bisect [https://www.kernel.org/pub/software/scm/git/docs/git-bisect-lk2009.html], Section G.3.9, “git-blame(1)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.9. git-blame(1)

2024-07-02

Revision History

2024-07-02

NAME

git-blame - Show what revision and author last modified each line of a file

SYNOPSIS

git blame [-c] [-b] [-l] [--root] [-t] [-f] [-n] [-s] [-e] [-p] [-w] [--incremental]
[-L <range>] [-S <revs-file>] [-M] [-C] [-C] [-C] [--since=<date>]
[--ignore-rev <rev>] [--ignore-revs-file <file>]
[--color-lines] [--color-by-age] [--progress] [--abbrev=<n>]
[--contents <file>] [<rev> | --reverse <rev>..<rev>] [--] <file>

DESCRIPTION

Annotates each line in the given file with information from the revision which last modified the line. Optionally, start annotating from the given revision.

When specified one or more times, *-L* restricts annotation to the requested lines.

The origin of lines is automatically followed across whole-file renames (currently there is no option to turn the rename-following off). To follow lines moved from one file to another, or to follow lines that were copied and pasted from another file, etc., see the *-C* and *-M* options.

The report does not tell you anything about lines which have been deleted or replaced; you need to use a tool such as *git diff* or the "pickaxe" interface briefly mentioned in the following paragraph.

Apart from supporting file annotation, Git also supports searching the development history for when a code snippet occurred in a change. This makes it possible to track when a code snippet was added to a file, moved or copied between files, and eventually deleted or replaced. It works by searching for a text string in the diff. A small example of the pickaxe interface that searches for *blame_usage*:

```
$ git log --pretty=oneline -S'blame_usage'
5040f17eba15504bad66b14a645bddd9b015ebb7 blame -S <ancestry-file>
ea4c7f9bf69e781dd0cd88d2bccb2bf5cc15c9a7 git-blame: Make the output
```

OPTIONS

-b

Show blank SHA-1 for boundary commits. This can also be controlled via the *blame.blankBoundary* config option.

--root

Do not treat root commits as boundaries. This can also be controlled via the *blame.showRoot* config option.

--show-stats

Include additional statistics at the end of blame output.

-L <start>,<end> , -L :<funcname>

Annotate only the line range given by *<start>*,*<end>*, or by the function name regex *<funcname>*. May be specified multiple times. Overlapping ranges are allowed.

<start> and *<end>* are optional. *-L <start>* or *-L <start>*, spans from *<start>* to end of file. *-L ,<end>* spans from start of file to *<end>*.

<start> and *<end>* can take one of these forms:

- number

  If *<start>* or *<end>* is a number, it specifies an absolute line number (lines count from 1).

- */regex/*

  This form will use the first line matching the given POSIX regex. If *<start>* is a regex, it will search from the end of the previous *-L* range, if any, otherwise from the start of file. If *<start>* is *^/regex/*, it will search from the start of file. If *<end>* is a regex, it will search starting at the line given by *<start>*.

- +offset or -offset

  This is only valid for *<end>* and will specify a number of lines before or after the line given by *<start>*.

If *:<funcname>* is given in place of *<start>* and *<end>*, it is a regular expression that denotes the range from the first funcname line that matches *<funcname>*, up to the next funcname line. *:<funcname>* searches from the end of the previous *-L* range, if any, otherwise from the start of file. *^:<funcname>* searches from the start of file. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

-l

Show long rev (Default: off).

-t

Show raw timestamp (Default: off).

-S <revs-file>

Use revisions from revs-file instead of calling Section G.3.120, “git-rev-list(1)”.

--reverse <rev>..<rev>

Walk history forward instead of backward. Instead of showing the revision in which a line appeared, this shows the last revision in which a line has existed. This requires a range of revision like START..END where

the path to blame exists in START. *git blame --reverse START* is taken as *git blame --reverse START..HEAD* for convenience.

--first-parent

Follow only the first parent commit upon seeing a merge commit. This option can be used to determine when a line was introduced to a particular integration branch, rather than when it was introduced to the history overall.

-p , --porcelain

Show in a format designed for machine consumption.

--line-porcelain

Show the porcelain format, but output commit information for each line, not just the first time a commit is referenced. Implies --porcelain.

--incremental

Show the result incrementally in a format designed for machine consumption.

--encoding=<encoding>

Specifies the encoding used to output author names and commit summaries. Setting it to *none* makes blame output unconverted data. For more information see the discussion about encoding in the Section G.3.74, "git-log(1)" manual page.

--contents <file>

Annotate using the contents from the named file, starting from <rev> if it is specified, and HEAD otherwise. You may specify - to make the command read from the standard input for the file contents.

--date <format>

Specifies the format used to output dates. If --date is not provided, the value of the blame.date config variable is used. If the blame.date config variable is also not set, the iso format is used. For supported values, see the discussion of the --date option at Section G.3.74, "git-log(1)".

--[no-]progress

Progress status is reported on the standard error stream by default when it is attached to a terminal. This flag enables progress reporting even if not attached to a terminal. Can't use *--progress* together with *--porcelain* or *--incremental*.

-M[<num>]

Detect moved or copied lines within a file. When a commit moves or copies a block of lines (e.g. the original file has A and then B, and the commit changes it to B and then A), the traditional *blame* algorithm notices only half of the movement and typically blames the lines that were moved up (i.e. B) to the parent and assigns blame to the lines that were moved down (i.e. A) to the child commit. With this option, both groups of lines are blamed on the parent by running extra passes of inspection.

<num> is optional but it is the lower bound on the number of alphanumeric characters that Git must detect as moving/copying within a file for it to associate those lines with the parent commit. The default value is 20.

-C[<num>]

In addition to *-M*, detect lines moved or copied from other files that were modified in the same commit. This is useful when you reorganize your program and move code around across files. When this option is given twice, the command additionally looks for copies from other files in the commit that creates the file. When this option is given three times, the command additionally looks for copies from other files in any commit.

<num> is optional but it is the lower bound on the number of alphanumeric characters that Git must detect as moving/copying between files for it to associate those lines with the parent commit. And the default value is 40. If there are more than one *-C* options given, the <num> argument of the last *-C* will take effect.

--ignore-rev <rev>

Ignore changes made by the revision when assigning blame, as if the change never happened. Lines that were changed or added by an ignored commit will be blamed on the previous commit that changed that line or nearby lines. This option may be specified multiple times to ignore more than one revision. If the *blame.markIgnoredLines* config option is set, then lines that were changed by an ignored commit and attributed to another commit will be marked with a *?* in the blame output. If the *blame.markUnblamableLines* config option is set, then those lines touched by an ignored commit that we could not attribute to another revision are marked with a *.

--ignore-revs-file <file>

Ignore revisions listed in *file*, which must be in the same format as an *fsck.skipList*. This option may be repeated, and these files will be processed after any files specified with the *blame.ignoreRevsFile* config option. An empty file name, *""*, will clear the list of revs from previously processed files.

--color-lines

Color line annotations in the default format differently if they come from the same commit as the preceding line. This makes it easier to distinguish code blocks introduced by different commits. The color defaults to cyan and can be adjusted using the *color.blame.repeatedLines* config option.

--color-by-age

Color line annotations depending on the age of the line in the default format. The *color.blame.highlightRecent* config option controls what color is used for each range of age.

-h

Show help message.

-c

Use the same output mode as Section G.3.4, “git-annotate(1)” (Default: off).

--score-debug

Include debugging information related to the movement of lines between files (see *-C*) and lines moved within a file (see *-M*). The first number listed is the score. This is the number of alphanumeric characters detected as having been moved between or within files. This must be above a certain threshold for *git blame* to consider those lines of code to have been moved.

-f , --show-name

Show the filename in the original commit. By default the filename is shown if there is any line that came from a file with a different name, due to rename detection.

-n , --show-number

Show the line number in the original commit (Default: off).

-s

Suppress the author name and timestamp from the output.

-e , --show-email

Show the author email instead of the author name (Default: off). This can also be controlled via the *blame.showEmail* config option.

-w

Ignore whitespace when comparing the parent's version and the child's to find where the lines came from.

--abbrev=<n>

Instead of using the default 7+1 hexadecimal digits as the abbreviated object name, use <m>+1 digits, where <m> is at least <n> but ensures the commit object names are unique. Note that 1 column is used for a caret to mark the boundary commit.

THE DEFAULT FORMAT

When neither *--porcelain* nor *--incremental* option is specified, *git blame* will output annotation for each line with:

- abbreviated object name for the commit the line came from;
- author ident (by default the author name and date, unless *-s* or *-e* is specified); and
- line number

before the line contents.

THE PORCELAIN FORMAT

In this format, each line is output after a header; the header at the minimum has the first line which has:

- 40-byte SHA-1 of the commit the line is attributed to;
- the line number of the line in the original file;
- the line number of the line in the final file;
- on a line that starts a group of lines from a different commit than the previous one, the number of lines in this group. On subsequent lines this field is absent.

This header line is followed by the following information at least once for each commit:

- the author name ("author"), email ("author-mail"), time ("author-time"), and time zone ("author-tz"); similarly for committer.
- the filename in the commit that the line is attributed to.
- the first line of the commit log message ("summary").

The contents of the actual line are output after the above header, prefixed by a TAB. This is to allow adding more header elements later.

The porcelain format generally suppresses commit information that has already been seen. For example, two lines that are blamed to the same commit will both be shown, but the details for that commit will be shown only once. This is more efficient, but may require more state be kept by the reader. The *--line-porcelain* option can be used to output full commit information for each line, allowing simpler (but less efficient) usage like:

```
# count the number of lines attributed to each author
git blame --line-porcelain file |
sed -n 's/^author //p' |
sort | uniq -c | sort -rn
```

SPECIFYING RANGES

Unlike *git blame* and *git annotate* in older versions of git, the extent of the annotation can be limited to both line ranges and revision ranges. The *-L* option, which limits annotation to a range of lines, may be specified multiple times.

When you are interested in finding the origin for lines 40-60 for file *foo*, you can use the *-L* option like so (they mean the same thing -- both ask for 21 lines starting at line 40):

```
git blame -L 40,60 foo
git blame -L 40,+21 foo
```

Also you can use a regular expression to specify the line range:

```
git blame -L '/^sub hello {/,/^}$/' foo
```

which limits the annotation to the body of the *hello* subroutine.

When you are not interested in changes older than version v2.6.18, or changes older than 3 weeks, you can use revision range specifiers similar to *git rev-list*:

```
git blame v2.6.18.. -- foo
git blame --since=3.weeks -- foo
```

When revision range specifiers are used to limit the annotation, lines that have not changed since the range boundary (either the commit v2.6.18 or the most recent commit that is more than 3 weeks old in the above example) are blamed for that range boundary commit.

A particularly useful way is to see if an added file has lines created by copy-and-paste from existing files. Sometimes this indicates that the developer was being sloppy and did not refactor the code properly. You can first find the commit that introduced the file with:

```
git log --diff-filter=A --pretty=short -- foo
```

and then annotate the change between the commit and its parents, using *commit^!* notation:

```
git blame -C -C -f $commit^! -- foo
```

INCREMENTAL OUTPUT

When called with *--incremental* option, the command outputs the result as it is built. The output generally will talk about lines touched by more recent commits first (i.e. the lines will be annotated out of order) and is meant to be used by interactive viewers.

The output format is similar to the Porcelain format, but it does not contain the actual lines from the file that is being annotated.

1. Each blame entry always starts with a line of:

   ```
   <40-byte-hex-sha1> <sourceline> <resultline> <num-lines>
   ```

   Line numbers count from 1.

2. The first time that a commit shows up in the stream, it has various other information about it printed out with a one-word tag at the beginning of each line describing the extra commit information (author, email, committer, dates, summary, etc.).

3. Unlike the Porcelain format, the filename information is always given and terminates the entry:

   ```
   "filename" <whitespace-quoted-filename-goes-here>
   ```

   and thus it is really quite easy to parse for some line- and word-oriented parser (which should be quite natural for most scripting languages).

> **Note**
>
> For people who do parsing: to make it more robust, just ignore any lines between the first and last one ("<sha1>" and "filename" lines) where you do not recognize the tag words (or care about

> that particular one) at the beginning of the "extended information" lines. That way, if there is ever added information (like the commit encoding or extended commit commentary), a blame viewer will not care.

MAPPING AUTHORS

See Section G.4.9, “gitmailmap(5)”.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

blame.blankBoundary

Show blank commit object name for boundary commits in Section G.3.9, “git-blame(1)”. This option defaults to false.

blame.coloring

This determines the coloring scheme to be applied to blame output. It can be *repeatedLines*, *highlightRecent*, or *none* which is the default.

blame.date

Specifies the format used to output dates in Section G.3.9, “git-blame(1)”. If unset the iso format is used. For supported values, see the discussion of the *--date* option at Section G.3.74, “git-log(1)”.

blame.showEmail

Show the author email instead of author name in Section G.3.9, “git-blame(1)”. This option defaults to false.

blame.showRoot

Do not treat root commits as boundaries in Section G.3.9, “git-blame(1)”. This option defaults to false.

blame.ignoreRevsFile

Ignore revisions listed in the file, one unabbreviated object name per line, in Section G.3.9, “git-blame(1)”. Whitespace and comments beginning with # are ignored. This option may be repeated multiple times. Empty file names will reset the list of ignored revisions. This option will be handled before the command line option *--ignore-revs-file*.

blame.markUnblamableLines

Mark lines that were changed by an ignored revision that we could not attribute to another commit with a * in the output of Section G.3.9, “git-blame(1)”.

blame.markIgnoredLines

Mark lines that were changed by an ignored revision that we attributed to another commit with a *?* in the output of Section G.3.9, “git-blame(1)”.

SEE ALSO

Section G.3.4, “git-annotate(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.10. git-branch(1)

2024-07-02

Revision History

2024-07-02

NAME

git-branch - List, create, or delete branches

SYNOPSIS

git branch [--color[=<when>] | --no-color] [--show-current]
[-v [--abbrev=<n> | --no-abbrev]]
[--column[=<options>] | --no-column] [--sort=<key>]
[--merged [<commit>]] [--no-merged [<commit>]]
[--contains [<commit>]] [--no-contains [<commit>]]
[--points-at <object>] [--format=<format>]
[(-r | --remotes) | (-a | --all)]
[--list] [<pattern>…]
git branch [--track[=(direct|inherit)] | --no-track] [-f]
[--recurse-submodules] <branchname> [<start-point>]
git branch (--set-upstream-to=<upstream> | -u <upstream>) [<branchname>]
git branch --unset-upstream [<branchname>]
git branch (-m | -M) [<oldbranch>] <newbranch>
git branch (-c | -C) [<oldbranch>] <newbranch>
git branch (-d | -D) [-r] <branchname>…
git branch --edit-description [<branchname>]

DESCRIPTION

If *--list* is given, or if there are no non-option arguments, existing branches are listed; the current branch will be highlighted in green and marked with an asterisk. Any branches checked out in linked worktrees will be highlighted in cyan and marked with a plus sign. Option *-r* causes the remote-tracking branches to be listed, and option *-a* shows both local and remote branches.

If a *<pattern>* is given, it is used as a shell wildcard to restrict the output to matching branches. If multiple patterns are given, a branch is shown if it matches any of the patterns.

Note that when providing a *<pattern>*, you must use *--list*; otherwise the command may be interpreted as branch creation.

With *--contains*, shows only the branches that contain the named commit (in other words, the branches whose tip commits are descendants of the named commit), *--no-contains* inverts it. With *--merged*, only branches merged into the named commit (i.e. the branches whose tip commits are reachable from the named commit) will be listed. With *--no-merged* only branches not merged into the named commit will be listed. If the <commit> argument is missing it defaults to *HEAD* (i.e. the tip of the current branch).

The command's second form creates a new branch head named <branchname> which points to the current *HEAD*, or <start-point> if given. As a special case, for <start-point>, you may use *"A...B"* as a shortcut for the merge base of *A* and *B* if there is exactly one merge base. You can leave out at most one of *A* and *B*, in which case it defaults to *HEAD*.

Note that this will create the new branch, but it will not switch the working tree to it; use "git switch <newbranch>" to switch to the new branch.

When a local branch is started off a remote-tracking branch, Git sets up the branch (specifically the *branch.<name>.remote* and *branch.<name>.merge* configuration entries) so that *git pull* will appropriately merge from the remote-tracking branch. This behavior may be changed via the global *branch.autoSetupMerge* configuration flag. That setting can be overridden by using the *--track* and *--no-track* options, and changed later using *git branch --set-upstream-to*.

With a *-m* or *-M* option, <oldbranch> will be renamed to <newbranch>. If <oldbranch> had a corresponding reflog, it is renamed to match <newbranch>, and a reflog entry is created to remember the branch renaming. If <newbranch> exists, -M must be used to force the rename to happen.

The *-c* and *-C* options have the exact same semantics as *-m* and *-M*, except instead of the branch being renamed, it will be copied to a new name, along with its config and reflog.

With a *-d* or *-D* option, *<branchname>* will be deleted. You may specify more than one branch for deletion. If the branch currently has a reflog then the reflog will also be deleted.

Use *-r* together with *-d* to delete remote-tracking branches. Note, that it only makes sense to delete remote-tracking branches if they no longer exist in the remote repository or if *git fetch* was configured not to fetch them again. See also the *prune* subcommand of Section G.3.112, “git-remote(1)” for a way to clean up all obsolete remote-tracking branches.

OPTIONS

-d , --delete

Delete a branch. The branch must be fully merged in its upstream branch, or in *HEAD* if no upstream was set with *--track* or *--set-upstream-to*.

-D

Shortcut for *--delete --force*.

--create-reflog

Create the branch's reflog. This activates recording of all changes made to the branch ref, enabling use of date based sha1 expressions such as "<branchname>@{yesterday}". Note that in non-bare repositories, reflogs are usually enabled by default by the *core.logAllRefUpdates* config option. The negated form *--no-create-reflog* only overrides an earlier *--create-reflog*, but currently does not negate the setting of *core.logAllRefUpdates*.

-f , --force

Reset <branchname> to <start-point>, even if <branchname> exists already. Without *-f*, *git branch* refuses to change an existing branch. In combination with *-d* (or *--delete*), allow deleting the branch irrespective of its merged status, or whether it even points to a valid commit. In combination with *-m* (or *--move*), allow renaming the branch even if the new branch name already exists, the same applies for *-c* (or *--copy*).

Note that *git branch -f <branchname> [<start-point>]*, even with *-f*, refuses to change an existing branch *<branchname>* that is checked out in another worktree linked to the same repository.

-m , --move

Move/rename a branch, together with its config and reflog.

-M

Shortcut for *--move --force*.

-c , --copy

Copy a branch, together with its config and reflog.

-C

Shortcut for *--copy --force*.

--color[=<when>]

Color branches to highlight current, local, and remote-tracking branches. The value must be always (the default), never, or auto.

--no-color

Turn off branch colors, even when the configuration file gives the default to color output. Same as *--color=never*.

-i , --ignore-case

Sorting and filtering branches are case insensitive.

--omit-empty

Do not print a newline after formatted refs where the format expands to the empty string.

--column[=<options>] , --no-column

Display branch listing in columns. See configuration variable *column.branch* for option syntax. *--column* and *--no-column* without options are equivalent to *always* and *never* respectively.

This option is only applicable in non-verbose mode.

-r , --remotes

List or delete (if used with -d) the remote-tracking branches. Combine with *--list* to match the optional pattern(s).

-a , --all

List both remote-tracking branches and local branches. Combine with *--list* to match optional pattern(s).

-l , --list

List branches. With optional *<pattern>...*, e.g. *git branch --list 'maint-*'*, list only the branches that match the pattern(s).

--show-current

Print the name of the current branch. In detached HEAD state, nothing is printed.

-v , -vv , --verbose

When in list mode, show sha1 and commit subject line for each head, along with relationship to upstream branch (if any). If given twice, print the path of the linked worktree (if any) and the name of the upstream branch, as well (see also *git remote show <remote>*). Note that the current worktree's HEAD will not have its path printed (it will always be your current directory).

-q , --quiet

Be more quiet when creating or deleting a branch, suppressing non-error messages.

--abbrev=<n>

In the verbose listing that show the commit object name, show the shortest prefix that is at least *<n>* hexdigits long that uniquely refers the object. The default value is 7 and can be overridden by the *core.abbrev* config option.

--no-abbrev

Display the full sha1s in the output listing rather than abbreviating them.

-t , --track[=(direct|inherit)]

When creating a new branch, set up *branch.<name>.remote* and *branch.<name>.merge* configuration entries to set "upstream" tracking configuration for the new branch. This configuration will tell git to show the

relationship between the two branches in *git status* and *git branch -v*. Furthermore, it directs *git pull* without arguments to pull from the upstream when the new branch is checked out.

The exact upstream branch is chosen depending on the optional argument: *-t*, *--track*, or *--track=direct* means to use the start-point branch itself as the upstream; *--track=inherit* means to copy the upstream configuration of the start-point branch.

The branch.autoSetupMerge configuration variable specifies how *git switch*, *git checkout* and *git branch* should behave when neither *--track* nor *--no-track* are specified:

The default option, *true*, behaves as though *--track=direct* were given whenever the start-point is a remote-tracking branch. *false* behaves as if *--no-track* were given. *always* behaves as though *--track=direct* were given. *inherit* behaves as though *--track=inherit* were given. *simple* behaves as though *--track=direct* were given only when the start-point is a remote-tracking branch and the new branch has the same name as the remote branch.

See Section G.3.102, “git-pull(1)” and Section G.3.29, “git-config(1)” for additional discussion on how the *branch.<name>.remote* and *branch.<name>.merge* options are used.

--no-track

Do not set up "upstream" configuration, even if the branch.autoSetupMerge configuration variable is set.

--recurse-submodules

THIS OPTION IS EXPERIMENTAL! Causes the current command to recurse into submodules if *submodule.propagateBranches* is enabled. See *submodule.propagateBranches* in Section G.3.29, “git-config(1)”. Currently, only branch creation is supported.

When used in branch creation, a new branch <branchname> will be created in the superproject and all of the submodules in the superproject's <start-point>. In submodules, the branch will point to the submodule commit in the superproject's <start-point> but the branch's tracking information will be set up based on the submodule's branches and remotes e.g. *git branch --recurse-submodules topic origin/main* will create the submodule branch "topic" that points to the submodule commit in the superproject's "origin/main", but tracks the submodule's "origin/main".

--set-upstream

As this option had confusing syntax, it is no longer supported. Please use *--track* or *--set-upstream-to* instead.

-u <upstream> , --set-upstream-to=<upstream>

Set up <branchname>'s tracking information so <upstream> is considered <branchname>'s upstream branch. If no <branchname> is specified, then it defaults to the current branch.

--unset-upstream

Remove the upstream information for <branchname>. If no branch is specified it defaults to the current branch.

--edit-description

Open an editor and edit the text to explain what the branch is for, to be used by various other commands (e.g. *format-patch*, *request-pull*, and *merge* (if enabled)). Multi-line explanations may be used.

--contains [<commit>]

Only list branches which contain the specified commit (HEAD if not specified). Implies *--list*.

--no-contains [<commit>]

Only list branches which don't contain the specified commit (HEAD if not specified). Implies *--list*.

--merged [<commit>]

Only list branches whose tips are reachable from the specified commit (HEAD if not specified). Implies *--list*.

--no-merged [<commit>]

Only list branches whose tips are not reachable from the specified commit (HEAD if not specified). Implies *--list*.

<branchname>

The name of the branch to create or delete. The new branch name must pass all checks defined by Section G.3.17, "git-check-ref-format(1)". Some of these checks may restrict the characters allowed in a branch name.

<start-point>

The new branch head will point to this commit. It may be given as a branch name, a commit-id, or a tag. If this option is omitted, the current HEAD will be used instead.

<oldbranch>

The name of an existing branch. If this option is omitted, the name of the current branch will be used instead.

<newbranch>

The new name for an existing branch. The same restrictions as for <branchname> apply.

--sort=<key>

Sort based on the key given. Prefix - to sort in descending order of the value. You may use the --sort=<key> option multiple times, in which case the last key becomes the primary key. The keys supported are the same as those in *git for-each-ref*. Sort order defaults to the value configured for the *branch.sort* variable if it exists, or to sorting based on the full refname (including *refs/*... prefix). This lists detached HEAD (if present) first, then local branches and finally remote-tracking branches. See Section G.3.29, "git-config(1)".

--points-at <object>

Only list branches of the given object.

--format <format>

A string that interpolates *%(fieldname)* from a branch ref being shown and the object it points at. The format is the same as that of Section G.3.52, "git-for-each-ref(1)".

CONFIGURATION

pager.branch is only respected when listing branches, i.e., when *--list* is used or implied. The default is to use a pager. See Section G.3.29, "git-config(1)".

Everything above this line in this section isn't included from the Section G.3.29, "git-config(1)" documentation. The content that follows is the same as what's found there:

branch.autoSetupMerge

Tells *git branch*, *git switch* and *git checkout* to set up new branches so that Section G.3.102, "git-pull(1)" will appropriately merge from the starting point branch. Note that even if this option is not set, this behavior can be chosen per-branch using the *--track* and *--no-track* options. The valid settings are: *false* -- no automatic setup is done; *true* -- automatic setup is done when the starting point is a remote-tracking branch; *always* -- automatic setup is done when the starting point is either a local branch or remote-tracking branch; *inherit* -- if the starting point has a tracking configuration, it is copied to the new branch; *simple* -- automatic setup is done only when the starting point is a remote-tracking branch and the new branch has the same name as the remote branch. This option defaults to true.

branch.autoSetupRebase

When a new branch is created with *git branch*, *git switch* or *git checkout* that tracks another branch, this variable tells Git to set up pull to rebase instead of merge (see "branch.<name>.rebase"). When *never*, rebase is never automatically set to true. When *local*, rebase is set to true for tracked branches of other local branches. When *remote*, rebase is set to true for tracked branches of remote-tracking branches. When *always*, rebase will be set to true for all tracking branches. See "branch.autoSetupMerge" for details on how to set up a branch to track another branch. This option defaults to never.

branch.sort

This variable controls the sort ordering of branches when displayed by Section G.3.10, “git-branch(1)”. Without the "--sort=<value>" option provided, the value of this variable will be used as the default. See Section G.3.52, “git-for-each-ref(1)” field names for valid values.

branch.<name>.remote

When on branch <name>, it tells *git fetch* and *git push* which remote to fetch from or push to. The remote to push to may be overridden with *remote.pushDefault* (for all branches). The remote to push to, for the current branch, may be further overridden by *branch.<name>.pushRemote*. If no remote is configured, or if you are not on any branch and there is more than one remote defined in the repository, it defaults to *origin* for fetching and *remote.pushDefault* for pushing. Additionally, . (a period) is the current local repository (a dot-repository), see *branch.<name>.merge*'s final note below.

branch.<name>.pushRemote

When on branch <name>, it overrides *branch.<name>.remote* for pushing. It also overrides *remote.pushDefault* for pushing from branch <name>. When you pull from one place (e.g. your upstream) and push to another place (e.g. your own publishing repository), you would want to set *remote.pushDefault* to specify the remote to push to for all branches, and use this option to override it for a specific branch.

branch.<name>.merge

Defines, together with branch.<name>.remote, the upstream branch for the given branch. It tells *git fetch*/*git pull*/*git rebase* which branch to merge and can also affect *git push* (see push.default). When in branch <name>, it tells *git fetch* the default refspec to be marked for merging in FETCH_HEAD. The value is handled like the remote part of a refspec, and must match a ref which is fetched from the remote given by "branch.<name>.remote". The merge information is used by *git pull* (which first calls *git fetch*) to lookup the default branch for merging. Without this option, *git pull* defaults to merge the first refspec fetched. Specify multiple values to get an octopus merge. If you wish to setup *git pull* so that it merges into <name> from another branch in the local repository, you can point branch.<name>.merge to the desired branch, and use the relative path setting . (a period) for branch.<name>.remote.

branch.<name>.mergeOptions

Sets default options for merging into branch <name>. The syntax and supported options are the same as those of Section G.3.86, “git-merge(1)”, but option values containing whitespace characters are currently not supported.

branch.<name>.rebase

When true, rebase the branch <name> on top of the fetched branch, instead of merging the default branch from the default remote when "git pull" is run. See "pull.rebase" for doing this in a non branch-specific manner.

When *merges* (or just *m*), pass the *--rebase-merges* option to *git rebase* so that the local merge commits are included in the rebase (see Section G.3.107, “git-rebase(1)” for details).

When the value is *interactive* (or just *i*), the rebase is run in interactive mode.

NOTE: this is a possibly dangerous operation; do **not** use it unless you understand the implications (see Section G.3.107, “git-rebase(1)” for details).

branch.<name>.description

Branch description, can be edited with *git branch --edit-description*. Branch description is automatically added to the format-patch cover letter or request-pull summary.

EXAMPLES

Start development from a known tag

```
$ git clone git://git.kernel.org/pub/scm/.../linux-2.6 my2.6
$ cd my2.6
$ git branch my2.6.14 v2.6.14   ❶
$ git switch my2.6.14
```

❶ This step and the next one could be combined into a single step with "checkout -b my2.6.14 v2.6.14".

Delete an unneeded branch

```
$ git clone git://git.kernel.org/.../git.git my.git
$ cd my.git
$ git branch -d -r origin/todo origin/html origin/man   ❶
$ git branch -D test                                    ❷
```

❶ Delete the remote-tracking branches "todo", "html" and "man". The next *fetch* or *pull* will create them again unless you configure them not to. See Section G.3.49, “git-fetch(1)”.

❷ Delete the "test" branch even if the "master" branch (or whichever branch is currently checked out) does not have all commits from the test branch.

Listing branches from a specific remote

```
$ git branch -r -l '<remote>/<pattern>'                 ❶
$ git for-each-ref 'refs/remotes/<remote>/<pattern>'    ❷
```

❶ Using *-a* would conflate <remote> with any local branches you happen to have been prefixed with the same <remote> pattern.

❷ *for-each-ref* can take a wide range of options. See Section G.3.52, “git-for-each-ref(1)”

Patterns will normally need quoting.

NOTES

If you are creating a branch that you want to switch to immediately, it is easier to use the "git switch" command with its *-c* option to do the same thing with a single command.

The options *--contains*, *--no-contains*, *--merged* and *--no-merged* serve four related but different purposes:

- *--contains* <*commit*> is used to find all branches which will need special attention if <commit> were to be rebased or amended, since those branches contain the specified <commit>.
- *--no-contains* <*commit*> is the inverse of that, i.e. branches that don't contain the specified <commit>.
- *--merged* is used to find all branches which can be safely deleted, since those branches are fully contained by HEAD.
- *--no-merged* is used to find branches which are candidates for merging into HEAD, since those branches are not fully contained by HEAD.

When combining multiple *--contains* and *--no-contains* filters, only references that contain at least one of the *--contains* commits and contain none of the *--no-contains* commits are shown.

When combining multiple *--merged* and *--no-merged* filters, only references that are reachable from at least one of the *--merged* commits and from none of the *--no-merged* commits are shown.

SEE ALSO

Section G.3.17, “git-check-ref-format(1)”, Section G.3.49, “git-fetch(1)”, Section G.3.112, “git-remote(1)”, *Understanding history: What is a branch?* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html#what-is-a-branch] in the Git User's Manual.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.11. git-bugreport(1)

2024-07-02

Revision History

2024-07-02

NAME

git-bugreport - Collect information for user to file a bug report

SYNOPSIS

git bugreport [(-o | --output-directory) <path>]
[(-s | --suffix) <format> | --no-suffix]
[--diagnose[=<mode>]]

DESCRIPTION

Collects information about the user's machine, Git client, and repository state, in addition to a form requesting information about the behavior the user observed, and stores it in a single text file which the user can then share, for example to the Git mailing list, in order to report an observed bug.

The following information is requested from the user:

- Reproduction steps
- Expected behavior
- Actual behavior

The following information is captured automatically:

- *git version --build-options*
- uname sysname, release, version, and machine strings
- Compiler-specific info string
- A list of enabled hooks
- $SHELL

Additional information may be gathered into a separate zip archive using the *--diagnose* option, and can be attached alongside the bugreport document to provide additional context to readers.

This tool is invoked via the typical Git setup process, which means that in some cases, it might not be able to launch - for example, if a relevant config file is unreadable. In this kind of scenario, it may be helpful to manually gather the kind of information listed above when manually asking for help.

OPTIONS

-o <path> , --output-directory <path>

Place the resulting bug report file in <*path*> instead of the current directory.

-s <format> , --suffix <format> , --no-suffix

Specify an alternate suffix for the bugreport name, to create a file named *git-bugreport-<formatted-suffix>*. This should take the form of a strftime(3) format string; the current local time will be used. *--no-suffix* disables the suffix and the file is just named *git-bugreport* without any disambiguation measure.

--no-diagnose , --diagnose[=<mode>]

Create a zip archive of supplemental information about the user's machine, Git client, and repository state. The archive is written to the same output directory as the bug report and is named *git-diagnostics-<formatted-suffix>*.

Without *mode* specified, the diagnostic archive will contain the default set of statistics reported by *git diagnose*. An optional *mode* value may be specified to change which information is included in the archive. See Section G.3.40, “git-diagnose(1)” for the list of valid values for *mode* and details about their usage.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.12. git-bundle(1)

2024-07-02
Revision History
2024-07-02

NAME

git-bundle - Move objects and refs by archive

SYNOPSIS

git bundle create [-q | --quiet | --progress]
[--version=<version>] <file> <git-rev-list-args>
git bundle verify [-q | --quiet] <file>
git bundle list-heads <file> [<refname>…]
git bundle unbundle [--progress] <file> [<refname>…]

DESCRIPTION

Create, unpack, and manipulate "bundle" files. Bundles are used for the "offline" transfer of Git objects without an active "server" sitting on the other side of the network connection.

They can be used to create both incremental and full backups of a repository, and to relay the state of the references in one repository to another.

Git commands that fetch or otherwise "read" via protocols such as *ssh://* and *https://* can also operate on bundle files. It is possible Section G.3.24, “git-clone(1)” a new repository from a bundle, to use Section G.3.49, “git-fetch(1)” to fetch from one, and to list the references contained within it with Section G.3.76, “git-ls-remote(1)”. There's no corresponding "write" support, i.e.a *git push* into a bundle is not supported.

See the "EXAMPLES" section below for examples of how to use bundles.

BUNDLE FORMAT

Bundles are *.pack* files (see Section G.3.96, “git-pack-objects(1)”) with a header indicating what references are contained within the bundle.

Like the packed archive format itself bundles can either be self-contained, or be created using exclusions. See the "OBJECT PREREQUISITES" section below.

Bundles created using revision exclusions are "thin packs" created using the *--thin* option to Section G.3.96, “git-pack-objects(1)”, and unbundled using the *--fix-thin* option to Section G.3.69, “git-index-pack(1)”.

There is no option to create a "thick pack" when using revision exclusions, and users should not be concerned about the difference. By using "thin packs", bundles created using exclusions are smaller in size. That they're "thin" under the hood is merely noted here as a curiosity, and as a reference to other documentation.

See Section G.5.1, “gitformat-bundle(5)” for more details and the discussion of "thin pack" in Section G.5.5, “gitformat-pack(5)” for further details.

OPTIONS

create [options] <file> <git-rev-list-args>

Used to create a bundle named *file*. This requires the *<git-rev-list-args>* arguments to define the bundle contents. *options* contains the options specific to the *git bundle create* subcommand. If *file* is -, the bundle is written to stdout.

verify <file>

Used to check that a bundle file is valid and will apply cleanly to the current repository. This includes checks on the bundle format itself as well as checking that the prerequisite commits exist and are fully linked in the current repository. Then, *git bundle* prints a list of missing commits, if any. Finally, information about additional capabilities, such as "object filter", is printed. See "Capabilities" in Section G.5.1, “gitformat-bundle(5)” for more information. The exit code is zero for success, but will be nonzero if the bundle file is invalid. If *file* is -, the bundle is read from stdin.

list-heads <file>

Lists the references defined in the bundle. If followed by a list of references, only references matching those given are printed out. If *file* is -, the bundle is read from stdin.

unbundle <file>

Passes the objects in the bundle to *git index-pack* for storage in the repository, then prints the names of all defined references. If a list of references is given, only references matching those in the list are printed. This command is really plumbing, intended to be called only by *git fetch*. If *file* is -, the bundle is read from stdin.

<git-rev-list-args>

A list of arguments, acceptable to *git rev-parse* and *git rev-list* (and containing a named ref, see SPECIFYING REFERENCES below), that specifies the specific objects and references to transport. For example, *master~10..master* causes the current master reference to be packaged along with all objects added since its 10th ancestor commit. There is no explicit limit to the number of references and objects that may be packaged.

[<refname>…]

A list of references used to limit the references reported as available. This is principally of use to *git fetch*, which expects to receive only those references asked for and not necessarily everything in the pack (in this case, *git bundle* acts like *git fetch-pack*).

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

--version=<version>

Specify the bundle version. Version 2 is the older format and can only be used with SHA-1 repositories; the newer version 3 contains capabilities that permit extensions. The default is the oldest supported format, based on the hash algorithm in use.

-q , --quiet

This flag makes the command not to report its progress on the standard error stream.

SPECIFYING REFERENCES

Revisions must be accompanied by reference names to be packaged in a bundle.

More than one reference may be packaged, and more than one set of prerequisite objects can be specified. The objects packaged are those not contained in the union of the prerequisites.

The *git bundle create* command resolves the reference names for you using the same rules as *git rev-parse --abbrev-ref=loose*. Each prerequisite can be specified explicitly (e.g. *^master~10*), or implicitly (e.g. *master~10..master*, *--since=10.days.ago master*).

All of these simple cases are OK (assuming we have a "master" and "next" branch):

```
$ git bundle create master.bundle master
$ echo master | git bundle create master.bundle --stdin
$ git bundle create master-and-next.bundle master next
$ (echo master; echo next) | git bundle create master-and-next.bundle --
stdin
```

And so are these (and the same but omitted *--stdin* examples):

```
$ git bundle create recent-master.bundle master~10..master
$ git bundle create recent-updates.bundle master~10..master next~5..next
```

A revision name or a range whose right-hand-side cannot be resolved to a reference is not accepted:

```
$ git bundle create HEAD.bundle $(git rev-parse HEAD)
fatal: Refusing to create empty bundle.
$ git bundle create master-yesterday.bundle master~10..master~5
fatal: Refusing to create empty bundle.
```

OBJECT PREREQUISITES

When creating bundles it is possible to create a self-contained bundle that can be unbundled in a repository with no common history, as well as providing negative revisions to exclude objects needed in the earlier parts of the history.

Feeding a revision such as *new* to *git bundle create* will create a bundle file that contains all the objects reachable from the revision *new*. That bundle can be unbundled in any repository to obtain a full history that leads to the revision *new*:

```
$ git bundle create full.bundle new
```

A revision range such as *old..new* will produce a bundle file that will require the revision *old* (and any objects reachable from it) to exist for the bundle to be "unbundle"-able:

```
$ git bundle create full.bundle old..new
```

A self-contained bundle without any prerequisites can be extracted into anywhere, even into an empty repository, or be cloned from (i.e., *new*, but not *old..new*).

It is okay to err on the side of caution, causing the bundle file to contain objects already in the destination, as these are ignored when unpacking at the destination.

If you want to match *git clone --mirror*, which would include your refs such as *refs/remotes/**, use *--all*. If you want to provide the same set of refs that a clone directly from the source repository would get, use *--branches --tags* for the *<git-rev-list-args>*.

The *git bundle verify* command can be used to check whether your recipient repository has the required prerequisite commits for a bundle.

EXAMPLES

Assume you want to transfer the history from a repository R1 on machine A to another repository R2 on machine B. For whatever reason, direct connection between A and B is not allowed, but we can move data from A to B via some mechanism (CD, email, etc.). We want to update R2 with development made on the branch master in R1.

To bootstrap the process, you can first create a bundle that does not have any prerequisites. You can use a tag to remember up to what commit you last processed, in order to make it easy to later update the other repository with an incremental bundle:

```
machineA$ cd R1
machineA$ git bundle create file.bundle master
machineA$ git tag -f lastR2bundle master
```

Then you transfer file.bundle to the target machine B. Because this bundle does not require any existing object to be extracted, you can create a new repository on machine B by cloning from it:

```
machineB$ git clone -b master /home/me/tmp/file.bundle R2
```

This will define a remote called "origin" in the resulting repository that lets you fetch and pull from the bundle. The $GIT_DIR/config file in R2 will have an entry like this:

```
[remote "origin"]
    url = /home/me/tmp/file.bundle
    fetch = refs/heads/*:refs/remotes/origin/*
```

To update the resulting mine.git repository, you can fetch or pull after replacing the bundle stored at /home/me/tmp/file.bundle with incremental updates.

After working some more in the original repository, you can create an incremental bundle to update the other repository:

```
machineA$ cd R1
machineA$ git bundle create file.bundle lastR2bundle..master
machineA$ git tag -f lastR2bundle master
```

You then transfer the bundle to the other machine to replace /home/me/tmp/file.bundle, and pull from it.

```
machineB$ cd R2
machineB$ git pull
```

If you know up to what commit the intended recipient repository should have the necessary objects, you can use that knowledge to specify the prerequisites, giving a cut-off point to limit the revisions and objects that go in the resulting bundle. The previous example used the lastR2bundle tag for this purpose, but you can use any other options that you would give to the Section G.3.74, “git-log(1)” command. Here are more examples:

You can use a tag that is present in both:

```
$ git bundle create mybundle v1.0.0..master
```

You can use a prerequisite based on time:

```
$ git bundle create mybundle --since=10.days master
```

You can use the number of commits:

```
$ git bundle create mybundle -10 master
```

You can run *git-bundle verify* to see if you can extract from a bundle that was created with a prerequisite:

```
$ git bundle verify mybundle
```

This will list what commits you must have in order to extract from the bundle and will error out if you do not have them.

A bundle from a recipient repository's point of view is just like a regular repository which it fetches or pulls from. You can, for example, map references when fetching:

```
$ git fetch mybundle master:localRef
```

You can also see what references it offers:

```
$ git ls-remote mybundle
```

FILE FORMAT

See Section G.5.1, “gitformat-bundle(5)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.13. git-cat-file(1)

2024-07-02
Revision History
2024-07-02

NAME

git-cat-file - Provide contents or details of repository objects

SYNOPSIS

git cat-file <type> <object>
git cat-file (-e | -p) <object>
git cat-file (-t | -s) [--allow-unknown-type] <object>
git cat-file (--textconv | --filters)
[<rev>:<path|tree-ish> | --path=<path|tree-ish> <rev>]
git cat-file (--batch | --batch-check | --batch-command) [--batch-all-objects]
[--buffer] [--follow-symlinks] [--unordered]
[--textconv | --filters] [-Z]

DESCRIPTION

Output the contents or other properties such as size, type or delta information of one or more objects.

This command can operate in two modes, depending on whether an option from the *--batch* family is specified.

In non-batch mode, the command provides information on an object named on the command line.

In batch mode, arguments are read from standard input.

OPTIONS

<object>

The name of the object to show. For a more complete list of ways to spell object names, see the "SPECIFYING REVISIONS" section in Section G.4.14, “gitrevisions(7)”.

-t

Instead of the content, show the object type identified by *<object>*.

-s

Instead of the content, show the object size identified by *<object>*. If used with *--use-mailmap* option, will show the size of updated object after replacing idents using the mailmap mechanism.

-e

Exit with zero status if *<object>* exists and is a valid object. If *<object>* is of an invalid format, exit with non-zero status and emit an error on stderr.

-p

Pretty-print the contents of *<object>* based on its type.

<type>

Typically this matches the real type of *<object>* but asking for a type that can trivially be dereferenced from the given *<object>* is also permitted. An example is to ask for a "tree" with *<object>* being a commit object that contains it, or to ask for a "blob" with *<object>* being a tag object that points at it.

--[no-]mailmap , --[no-]use-mailmap

Use mailmap file to map author, committer and tagger names and email addresses to canonical real names and email addresses. See Section G.3.130, “git-shortlog(1)”.

--textconv

Show the content as transformed by a textconv filter. In this case, *<object>* has to be of the form *<tree-ish>:<path>*, or *:<path>* in order to apply the filter to the content recorded in the index at *<path>*.

--filters

Show the content as converted by the filters configured in the current working tree for the given *<path>* (i.e. smudge filters, end-of-line conversion, etc). In this case, *<object>* has to be of the form *<tree-ish>:<path>*, or *:<path>*.

--path=<path>

For use with *--textconv* or *--filters*, to allow specifying an object name and a path separately, e.g. when it is difficult to figure out the revision from which the blob came.

--batch , --batch=<format>

Print object information and contents for each object provided on stdin. May not be combined with any other options or arguments except *--textconv*, *--filters*, or *--use-mailmap*.

- When used with *--textconv* or *--filters*, the input lines must specify the path, separated by whitespace. See the section *BATCH OUTPUT* below for details.
- When used with *--use-mailmap*, for commit and tag objects, the contents part of the output shows the identities replaced using the mailmap mechanism, while the information part of the output shows the size of the object as if it actually recorded the replacement identities.

--batch-check , --batch-check=<format>

Print object information for each object provided on stdin. May not be combined with any other options or arguments except *--textconv*, *--filters* or *--use-mailmap*.

- When used with *--textconv* or *--filters*, the input lines must specify the path, separated by whitespace. See the section *BATCH OUTPUT* below for details.
- When used with *--use-mailmap*, for commit and tag objects, the printed object information shows the size of the object as if the identities recorded in it were replaced by the mailmap mechanism.

--batch-command , --batch-command=<format>

Enter a command mode that reads commands and arguments from stdin. May only be combined with *--buffer*, *--textconv*, *--use-mailmap* or *--filters*.

- When used with *--textconv* or *--filters*, the input lines must specify the path, separated by whitespace. See the section *BATCH OUTPUT* below for details.
- When used with *--use-mailmap*, for commit and tag objects, the *contents* command shows the identities replaced using the mailmap mechanism, while the *info* command shows the size of the object as if it actually recorded the replacement identities.

--batch-command recognizes the following commands:

contents <object>

Print object contents for object reference *<object>*. This corresponds to the output of *--batch*.

info <object>

Print object info for object reference *<object>*. This corresponds to the output of *--batch-check*.

flush

Used with *--buffer* to execute all preceding commands that were issued since the beginning or since the last flush was issued. When *--buffer* is used, no output will come until a *flush* is issued. When *--buffer* is not used, commands are flushed each time without issuing *flush*.

--batch-all-objects

Instead of reading a list of objects on stdin, perform the requested batch operation on all objects in the repository and any alternate object stores (not just reachable objects). Requires *--batch* or *--batch-check* be specified. By default, the objects are visited in order sorted by their hashes; see also *--unordered* below. Objects are presented as-is, without respecting the "replace" mechanism of Section G.3.114, “git-replace(1)”.

--buffer

Normally batch output is flushed after each object is output, so that a process can interactively read and write from *cat-file*. With this option, the output uses normal stdio buffering; this is much more efficient when invoking *--batch-check* or *--batch-command* on a large number of objects.

--unordered

When *--batch-all-objects* is in use, visit objects in an order which may be more efficient for accessing the object contents than hash order. The exact details of the order are unspecified, but if you do not require a specific order, this should generally result in faster output, especially with *--batch*. Note that *cat-file* will still show each object only once, even if it is stored multiple times in the repository.

--allow-unknown-type

Allow *-s* or *-t* to query broken/corrupt objects of unknown type.

--follow-symlinks

With *--batch* or *--batch-check*, follow symlinks inside the repository when requesting objects with extended SHA-1 expressions of the form tree-ish:path-in-tree. Instead of providing output about the link itself, provide output about the linked-to object. If a symlink points outside the tree-ish (e.g. a link to */foo* or a root-level link to *../foo*), the portion of the link which is outside the tree will be printed.

This option does not (currently) work correctly when an object in the index is specified (e.g. *:link* instead of *HEAD:link*) rather than one in the tree.

This option cannot (currently) be used unless *--batch* or *--batch-check* is used.

For example, consider a git repository containing:

```
f: a file containing "hello\n"
link: a symlink to f
dir/link: a symlink to ../f
plink: a symlink to ../f
alink: a symlink to /etc/passwd
```

For a regular file *f*, *echo HEAD:f | git cat-file --batch* would print

```
ce013625030ba8dba906f756967f9e9ca394464a blob 6
```

And *echo HEAD:link | git cat-file --batch --follow-symlinks* would print the same thing, as would *HEAD:dir/link*, as they both point at *HEAD:f*.

Without *--follow-symlinks*, these would print data about the symlink itself. In the case of *HEAD:link*, you would see

```
4d1ae35ba2c8ec712fa2a379db44ad639ca277bd blob 1
```

Both *plink* and *alink* point outside the tree, so they would respectively print:

```
symlink 4
../f
```

```
symlink 11
/etc/passwd
```

-Z

Only meaningful with *--batch*, *--batch-check*, or *--batch-command*; input and output is NUL-delimited instead of newline-delimited.

-z

Only meaningful with *--batch*, *--batch-check*, or *--batch-command*; input is NUL-delimited instead of newline-delimited. This option is deprecated in favor of *-Z* as the output can otherwise be ambiguous.

OUTPUT

If *-t* is specified, one of the <*type*>.

If *-s* is specified, the size of the <*object*> in bytes.

If *-e* is specified, no output, unless the <*object*> is malformed.

If *-p* is specified, the contents of <*object*> are pretty-printed.

If <*type*> is specified, the raw (though uncompressed) contents of the <*object*> will be returned.

BATCH OUTPUT

If *--batch* or *--batch-check* is given, *cat-file* will read objects from stdin, one per line, and print information about them. By default, the whole line is considered as an object, as if it were fed to Section G.3.121, “git-rev-parse(1)”.

When *--batch-command* is given, *cat-file* will read commands from stdin, one per line, and print information based on the command given. With *--batch-command*, the *info* command followed by an object will print information about the object the same way *--batch-check* would, and the *contents* command followed by an object prints contents in the same way *--batch* would.

You can specify the information shown for each object by using a custom <*format*>. The <*format*> is copied literally to stdout for each object, with placeholders of the form *%(atom)* expanded, followed by a newline. The available atoms are:

objectname

The full hex representation of the object name.

objecttype

The type of the object (the same as *cat-file -t* reports).

objectsize

The size, in bytes, of the object (the same as *cat-file -s* reports).

objectsize:disk

The size, in bytes, that the object takes up on disk. See the note about on-disk sizes in the *CAVEATS* section below.

deltabase

If the object is stored as a delta on-disk, this expands to the full hex representation of the delta base object name. Otherwise, expands to the null OID (all zeroes). See *CAVEATS* below.

rest

If this atom is used in the output string, input lines are split at the first whitespace boundary. All characters before that whitespace are considered to be the object name; characters after that first run of whitespace (i.e., the "rest" of the line) are output in place of the *%(rest)* atom.

If no format is specified, the default format is *%(objectname) %(objecttype) %(objectsize)*.

If *--batch* is specified, or if *--batch-command* is used with the *contents* command, the object information is followed by the object contents (consisting of *%(objectsize)* bytes), followed by a newline.

For example, *--batch* without a custom format would produce:

```
<oid> SP <type> SP <size> LF
<contents> LF
```

Whereas *--batch-check='%(objectname) %(objecttype)'* would produce:

```
<oid> SP <type> LF
```

If a name is specified on stdin that cannot be resolved to an object in the repository, then *cat-file* will ignore any custom format and print:

```
<object> SP missing LF
```

If a name is specified that might refer to more than one object (an ambiguous short sha), then *cat-file* will ignore any custom format and print:

```
<object> SP ambiguous LF
```

If *--follow-symlinks* is used, and a symlink in the repository points outside the repository, then *cat-file* will ignore any custom format and print:

```
symlink SP <size> LF
<symlink> LF
```

The symlink will either be absolute (beginning with a */*), or relative to the tree root. For instance, if dir/link points to *../../foo*, then *<symlink>* will be *../foo*. *<size>* is the size of the symlink in bytes.

If *--follow-symlinks* is used, the following error messages will be displayed:

```
<object> SP missing LF
```

is printed when the initial symlink requested does not exist.

```
dangling SP <size> LF
<object> LF
```

is printed when the initial symlink exists, but something that it (transitive-of) points to does not.

```
loop SP <size> LF
<object> LF
```

is printed for symlink loops (or any symlinks that require more than 40 link resolutions to resolve).

```
notdir SP <size> LF
<object> LF
```

is printed when, during symlink resolution, a file is used as a directory name.

Alternatively, when *-Z* is passed, the line feeds in any of the above examples are replaced with NUL terminators. This ensures that output will be parsable if the output itself would contain a linefeed and is thus recommended for scripting purposes.

CAVEATS

Note that the sizes of objects on disk are reported accurately, but care should be taken in drawing conclusions about which refs or objects are responsible for disk usage. The size of a packed non-delta object may be much larger than the size of objects which delta against it, but the choice of which object is the base and which is the delta is arbitrary and is subject to change during a repack.

Note also that multiple copies of an object may be present in the object database; in this case, it is undefined which copy's size or delta base will be reported.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.14. git-check-attr(1)

2024-07-02
Revision History
2024-07-02

NAME

git-check-attr - Display gitattributes information

SYNOPSIS

git check-attr [--source <tree-ish>] [-a | --all | <attr>…] [--] <pathname>…
git check-attr --stdin [-z] [--source <tree-ish>] [-a | --all | <attr>…]

DESCRIPTION

For every pathname, this command will list if each attribute is *unspecified*, *set*, or *unset* as a gitattribute on that pathname.

OPTIONS

-a, --all

List all attributes that are associated with the specified paths. If this option is used, then *unspecified* attributes will not be included in the output.

--cached

Consider *.gitattributes* in the index only, ignoring the working tree.

--stdin

Read pathnames from the standard input, one per line, instead of from the command line.

-z

The output format is modified to be machine-parsable. If *--stdin* is also given, input paths are separated with a NUL character instead of a linefeed character.

--source=<tree-ish>

Check attributes against the specified tree-ish. It is common to specify the source tree by naming a commit, branch, or tag associated with it.

--

Interpret all preceding arguments as attributes and all following arguments as path names.

If none of *--stdin*, *--all*, or -- is used, the first argument will be treated as an attribute and the rest of the arguments as pathnames.

OUTPUT

The output is of the form: <path> COLON SP <attribute> COLON SP <info> LF

unless *-z* is in effect, in which case NUL is used as delimiter: <path> NUL <attribute> NUL <info> NUL

<path> is the path of a file being queried, <attribute> is an attribute being queried, and <info> can be either:

unspecified

when the attribute is not defined for the path.

unset

when the attribute is defined as false.

set

when the attribute is defined as true.

<value>

when a value has been assigned to the attribute.

Buffering happens as documented under the *GIT_FLUSH* option in Section G.3.1, “git(1)”. The caller is responsible for avoiding deadlocks caused by overfilling an input buffer or reading from an empty output buffer.

EXAMPLES

In the examples, the following *.gitattributes* file is used:

```
*.java diff=java -crlf myAttr
NoMyAttr.java !myAttr
README caveat=unspecified
```

- Listing a single attribute:

```
$ git check-attr diff org/example/MyClass.java
```

```
org/example/MyClass.java: diff: java
```

- Listing multiple attributes for a file:

```
$ git check-attr crlf diff myAttr -- org/example/MyClass.java
org/example/MyClass.java: crlf: unset
org/example/MyClass.java: diff: java
org/example/MyClass.java: myAttr: set
```

- Listing all attributes for a file:

```
$ git check-attr --all -- org/example/MyClass.java
org/example/MyClass.java: diff: java
org/example/MyClass.java: myAttr: set
```

- Listing an attribute for multiple files:

```
$ git check-attr myAttr -- org/example/MyClass.java org/example/
NoMyAttr.java
org/example/MyClass.java: myAttr: set
org/example/NoMyAttr.java: myAttr: unspecified
```

- Not all values are equally unambiguous:

```
$ git check-attr caveat README
README: caveat: unspecified
```

SEE ALSO

Section G.4.2, “gitattributes(5)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.15. git-check-ignore(1)

2024-07-02

Revision History

2024-07-02

NAME

git-check-ignore - Debug gitignore / exclude files

SYNOPSIS

git check-ignore [<options>] <pathname>…
git check-ignore [<options>] --stdin

DESCRIPTION

For each pathname given via the command-line or from a file via *--stdin*, check whether the file is excluded by .gitignore (or other input files to the exclude mechanism) and output the path if it is excluded.

By default, tracked files are not shown at all since they are not subject to exclude rules; but see --no-index.

OPTIONS

-q, --quiet

Don't output anything, just set exit status. This is only valid with a single pathname.

-v, --verbose

Instead of printing the paths that are excluded, for each path that matches an exclude pattern, print the exclude pattern together with the path. (Matching an exclude pattern usually means the path is excluded, but if the pattern begins with "*!*" then it is a negated pattern and matching it means the path is NOT excluded.)

For precedence rules within and between exclude sources, see Section G.4.5, “gitignore(5)”.

--stdin

Read pathnames from the standard input, one per line, instead of from the command-line.

-z

The output format is modified to be machine-parsable (see below). If *--stdin* is also given, input paths are separated with a NUL character instead of a linefeed character.

-n, --non-matching

Show given paths which don't match any pattern. This only makes sense when *--verbose* is enabled, otherwise it would not be possible to distinguish between paths which match a pattern and those which don't.

--no-index

Don't look in the index when undertaking the checks. This can be used to debug why a path became tracked by e.g. *git add .* and was not ignored by the rules as expected by the user or when developing patterns including negation to match a path previously added with *git add -f*.

OUTPUT

By default, any of the given pathnames which match an ignore pattern will be output, one per line. If no pattern matches a given path, nothing will be output for that path; this means that path will not be ignored.

If *--verbose* is specified, the output is a series of lines of the form:

<source> <COLON> <linenum> <COLON> <pattern> <HT> <pathname>

<pathname> is the path of a file being queried, <pattern> is the matching pattern, <source> is the pattern's source file, and <linenum> is the line number of the pattern within that source. If the pattern contained a "*!*" prefix or "/" suffix, it will be preserved in the output. <source> will be an absolute path when referring to the file configured by *core.excludesFile*, or relative to the repository root when referring to *.git/info/exclude* or a per-directory exclude file.

If *-z* is specified, the pathnames in the output are delimited by the null character; if *--verbose* is also specified then null characters are also used instead of colons and hard tabs:

<source> <NULL> <linenum> <NULL> <pattern> <NULL> <pathname> <NULL>

If *-n* or *--non-matching* are specified, non-matching pathnames will also be output, in which case all fields in each output record except for <pathname> will be empty. This can be useful when running non-interactively, so that files can be incrementally streamed to STDIN of a long-running check-ignore process, and for each of these files, STDOUT will indicate whether that file matched a pattern or not. (Without this option, it would be impossible to tell whether the absence of output for a given file meant that it didn't match any pattern, or that the output hadn't been generated yet.)

Buffering happens as documented under the *GIT_FLUSH* option in Section G.3.1, “git(1)”. The caller is responsible for avoiding deadlocks caused by overfilling an input buffer or reading from an empty output buffer.

EXIT STATUS

0

One or more of the provided paths is ignored.

1

None of the provided paths are ignored.

128

A fatal error was encountered.

SEE ALSO

Section G.4.5, “gitignore(5)” Section G.3.29, “git-config(1)” Section G.3.75, “git-ls-files(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.16. git-check-mailmap(1)

2024-07-02
Revision History
2024-07-02

NAME

git-check-mailmap - Show canonical names and email addresses of contacts

SYNOPSIS

git check-mailmap [<options>] <contact>…

DESCRIPTION

For each Name <user@host> or <user@host> from the command-line or standard input (when using *--stdin*), look up the person's canonical name and email address (see "Mapping Authors" below). If found, print them; otherwise print the input as-is.

OPTIONS

--stdin

Read contacts, one per line, from the standard input after exhausting contacts provided on the command-line.

OUTPUT

For each contact, a single line is output, terminated by a newline. If the name is provided or known to the *mailmap*, Name <user@host> is printed; otherwise only <user@host> is printed.

CONFIGURATION

See *mailmap.file* and *mailmap.blob* in Section G.3.29, “git-config(1)” for how to specify a custom *.mailmap* target file or object.

MAPPING AUTHORS

See Section G.4.9, “gitmailmap(5)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.17. git-check-ref-format(1)

2024-07-02

Revision History

2024-07-02

NAME

git-check-ref-format - Ensures that a reference name is well formed

SYNOPSIS

git check-ref-format [--normalize]
[--[no-]allow-onelevel] [--refspec-pattern]
<refname>
git check-ref-format --branch <branchname-shorthand>

DESCRIPTION

Checks if a given *refname* is acceptable, and exits with a non-zero status if it is not.

A reference is used in Git to specify branches and tags. A branch head is stored in the *refs/heads* hierarchy, while a tag is stored in the *refs/tags* hierarchy of the ref namespace (typically in *$GIT_DIR/refs/heads* and *$GIT_DIR/refs/tags* directories or, as entries in file *$GIT_DIR/packed-refs* if refs are packed by *git gc*).

Git imposes the following rules on how references are named:

1. They can include slash / for hierarchical (directory) grouping, but no slash-separated component can begin with a dot . or end with the sequence *.lock*.
2. They must contain at least one /. This enforces the presence of a category like *heads/*, *tags/* etc. but the actual names are not restricted. If the *--allow-onelevel* option is used, this rule is waived.
3. They cannot have two consecutive dots .. anywhere.
4. They cannot have ASCII control characters (i.e. bytes whose values are lower than \040, or \177 *DEL*), space, tilde ~, caret ^, or colon *:* anywhere.
5. They cannot have question-mark *?*, asterisk *, or open bracket *[* anywhere. See the *--refspec-pattern* option below for an exception to this rule.
6. They cannot begin or end with a slash / or contain multiple consecutive slashes (see the *--normalize* option below for an exception to this rule).
7. They cannot end with a dot ..
8. They cannot contain a sequence *@{*.
9. They cannot be the single character @.
10. They cannot contain a \.

These rules make it easy for shell script based tools to parse reference names, pathname expansion by the shell when a reference name is used unquoted (by mistake), and also avoid ambiguities in certain reference name expressions (see Section G.4.14, “gitrevisions(7)”):

1. A double-dot .. is often used as in *ref1..ref2*, and in some contexts this notation means *^ref1 ref2* (i.e. not in *ref1* and in *ref2*).
2. A tilde ~ and caret ^ are used to introduce the postfix *nth parent* and *peel onion* operation.
3. A colon *:* is used as in *srcref:dstref* to mean "use srcref's value and store it in dstref" in fetch and push operations. It may also be used to select a specific object such as with *git cat-file*: "git cat-file blob v1.3.3:refs.c".
4. at-open-brace *@{* is used as a notation to access a reflog entry.

With the *--branch* option, the command takes a name and checks if it can be used as a valid branch name (e.g. when creating a new branch). But be cautious when using the previous checkout syntax that may refer to a detached HEAD state. The rule *git check-ref-format --branch $name* implements may be stricter than what *git check-ref-format refs/heads/$name* says (e.g. a dash may appear at the beginning of a ref component, but it is explicitly forbidden at the beginning of a branch name). When run with the *--branch* option in a repository, the input is first expanded for the previous checkout syntax *@{-n}*. For example, *@{-1}* is a way to refer the last thing that was checked out using "git switch" or "git checkout" operation. This option should be used by porcelains to accept this syntax anywhere a branch name is expected, so they can act as if you typed the branch name. As an exception note that, the previous checkout operation might result in a commit object name when the N-th last thing checked out was not a branch.

OPTIONS

--[no-]allow-onelevel

Controls whether one-level refnames are accepted (i.e., refnames that do not contain multiple /-separated components). The default is *--no-allow-onelevel*.

--refspec-pattern

Interpret <refname> as a reference name pattern for a refspec (as used with remote repositories). If this option is enabled, <refname> is allowed to contain a single * in the refspec (e.g., *foo/bar*/baz* or *foo/bar*baz/* but not *foo/bar*/baz**).

--normalize

Normalize *refname* by removing any leading slash (/) characters and collapsing runs of adjacent slashes between name components into a single slash. If the normalized refname is valid then print it to standard output and exit with a status of 0, otherwise exit with a non-zero status. (*--print* is a deprecated way to spell *--normalize*.)

EXAMPLES

- Print the name of the previous thing checked out:

```
$ git check-ref-format --branch @{-1}
```

- Determine the reference name to use for a new branch:

```
$ ref=$(git check-ref-format --normalize "refs/heads/$newbranch")||
{ echo "we do not like '$newbranch' as a branch name." >&2 ; exit 1 ; }
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.18. git-checkout-index(1)

2024-07-02
Revision History
2024-07-02

NAME

git-checkout-index - Copy files from the index to the working tree

SYNOPSIS

git checkout-index [-u] [-q] [-a] [-f] [-n] [--prefix=<string>]
[--stage=<number>|all]
[--temp]

[--ignore-skip-worktree-bits]
[-z] [--stdin]
[--] [<file>…]

DESCRIPTION

Copies all listed files from the index to the working directory (not overwriting existing files).

OPTIONS

-u , --index

update stat information for the checked out entries in the index file.

-q , --quiet

be quiet if files exist or are not in the index

-f , --force

forces overwrite of existing files

-a , --all

checks out all files in the index except for those with the skip-worktree bit set (see *--ignore-skip-worktree-bits*). Cannot be used together with explicit filenames.

-n , --no-create

Don't checkout new files, only refresh files already checked out.

--prefix=<string>

When creating files, prepend <string> (usually a directory including a trailing /)

--stage=<number>|all

Instead of checking out unmerged entries, copy out the files from the named stage. <number> must be between 1 and 3. Note: --stage=all automatically implies --temp.

--temp

Instead of copying the files to the working directory, write the content to temporary files. The temporary name associations will be written to stdout.

--ignore-skip-worktree-bits

Check out all files, including those with the skip-worktree bit set.

--stdin

Instead of taking a list of paths from the command line, read the list of paths from the standard input. Paths are separated by LF (i.e. one path per line) by default.

-z

Only meaningful with *--stdin*; paths are separated with NUL character instead of LF.

--

Do not interpret any more arguments as options.

The order of the flags used to matter, but not anymore.

Just doing *git checkout-index* does nothing. You probably meant *git checkout-index -a*. And if you want to force it, you want *git checkout-index -f -a*.

Intuitiveness is not the goal here. Repeatability is. The reason for the "no arguments means no work" behavior is that from scripts you are supposed to be able to do:

```
$ find . -name '*.h' -print0 | xargs -0 git checkout-index -f --
```

which will force all existing **.h* files to be replaced with their cached copies. If an empty command line implied "all", then this would force-refresh everything in the index, which was not the point. But since *git checkout-index* accepts --stdin it would be faster to use:

```
$ find . -name '*.h' -print0 | git checkout-index -f -z --stdin
```

The -- is just a good idea when you know the rest will be filenames; it will prevent problems with a filename of, for example, *-a*. Using -- is probably a good policy in scripts.

Using --temp or --stage=all

When *--temp* is used (or implied by *--stage=all*) *git checkout-index* will create a temporary file for each index entry being checked out. The index will not be updated with stat information. These options can be useful if the caller needs all stages of all unmerged entries so that the unmerged files can be processed by an external merge tool.

A listing will be written to stdout providing the association of temporary file names to tracked path names. The listing format has two variations:

1. tempname TAB path RS

   The first format is what gets used when *--stage* is omitted or is not *--stage=all*. The field tempname is the temporary file name holding the file content and path is the tracked path name in the index. Only the requested entries are output.

2. stage1temp SP stage2temp SP stage3tmp TAB path RS

   The second format is what gets used when *--stage=all*. The three stage temporary fields (stage1temp, stage2temp, stage3temp) list the name of the temporary file if there is a stage entry in the index or . if there is no stage entry. Paths which only have a stage 0 entry will always be omitted from the output.

In both formats RS (the record separator) is newline by default but will be the null byte if -z was passed on the command line. The temporary file names are always safe strings; they will never contain directory separators or whitespace characters. The path field is always relative to the current directory and the temporary file names are always relative to the top level directory.

If the object being copied out to a temporary file is a symbolic link the content of the link will be written to a normal file. It is up to the end-user or the Porcelain to make use of this information.

EXAMPLES

To update and refresh only the files already checked out

```
$ git checkout-index -n -f -a && git update-index --ignore-missing --
refresh
```

Using *git checkout-index* to "export an entire tree"

The prefix ability basically makes it trivial to use *git checkout-index* as an "export as tree" function. Just read the desired tree into the index, and do:

```
$ git checkout-index --prefix=git-export-dir/ -a
```

git checkout-index will "export" the index into the specified directory.

The final "/" is important. The exported name is literally just prefixed with the specified string. Contrast this with the following example.

Export files with a prefix

```
$ git checkout-index --prefix=.merged- Makefile
```

This will check out the currently cached copy of *Makefile* into the file *.merged-Makefile*.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.19. git-checkout(1)

2024-07-02
Revision History
2024-07-02

NAME

git-checkout - Switch branches or restore working tree files

SYNOPSIS

git checkout [-q] [-f] [-m] [<branch>]
git checkout [-q] [-f] [-m] --detach [<branch>]
git checkout [-q] [-f] [-m] [--detach] <commit>
git checkout [-q] [-f] [-m] [[-b|-B|--orphan] <new-branch>] [<start-point>]
git checkout [-f] <tree-ish> [--] <pathspec>…
git checkout [-f] <tree-ish> --pathspec-from-file=<file> [--pathspec-file-nul]
git checkout [-f|--ours|--theirs|-m|--conflict=<style>] [--] <pathspec>…
git checkout [-f|--ours|--theirs|-m|--conflict=<style>] --pathspec-from-file=<file> [--pathspec-file-nul]
git checkout (-p|--patch) [<tree-ish>] [--] [<pathspec>…]

DESCRIPTION

Updates files in the working tree to match the version in the index or the specified tree. If no pathspec was given, *git checkout* will also update *HEAD* to set the specified branch as the current branch.

git checkout [<branch>]

To prepare for working on <*branch*>, switch to it by updating the index and the files in the working tree, and by pointing *HEAD* at the branch. Local modifications to the files in the working tree are kept, so that they can be committed to the <*branch*>.

If <*branch*> is not found but there does exist a tracking branch in exactly one remote (call it <*remote*>) with a matching name and *--no-guess* is not specified, treat as equivalent to

```
$ git checkout -b <branch> --track <remote>/<branch>
```

You could omit <*branch*>, in which case the command degenerates to "check out the current branch", which is a glorified no-op with rather expensive side-effects to show only the tracking information, if it exists, for the current branch.

git checkout -b|-B <new-branch> [<start-point>]

Specifying *-b* causes a new branch to be created as if Section G.3.10, “git-branch(1)” were called and then checked out. In this case you can use the *--track* or *--no-track* options, which will be passed to *git branch*. As a convenience, *--track* without *-b* implies branch creation; see the description of *--track* below.

If *-B* is given, *<new-branch>* is created if it doesn't exist; otherwise, it is reset. This is the transactional equivalent of

```
$ git branch -f <branch> [<start-point>]
$ git checkout <branch>
```

that is to say, the branch is not reset/created unless "git checkout" is successful (e.g., when the branch is in use in another worktree, not just the current branch stays the same, but the branch is not reset to the start-point, either).

git checkout --detach [<branch>] , *git checkout* [--detach] <commit>

Prepare to work on top of *<commit>*, by detaching *HEAD* at it (see "DETACHED HEAD" section), and updating the index and the files in the working tree. Local modifications to the files in the working tree are kept, so that the resulting working tree will be the state recorded in the commit plus the local modifications.

When the *<commit>* argument is a branch name, the *--detach* option can be used to detach *HEAD* at the tip of the branch (*git checkout <branch>* would check out that branch without detaching *HEAD*).

Omitting *<branch>* detaches *HEAD* at the tip of the current branch.

git checkout [-f|--ours|--theirs|-m|--conflict=<style>] [<tree-ish>] [--] <pathspec>… , *git checkout* [-f|--ours|--theirs|-m|--conflict=<style>] [<tree-ish>] --pathspec-from-file=<file> [--pathspec-file-nul]

Overwrite the contents of the files that match the pathspec. When the *<tree-ish>* (most often a commit) is not given, overwrite working tree with the contents in the index. When the *<tree-ish>* is given, overwrite both the index and the working tree with the contents at the *<tree-ish>*.

The index may contain unmerged entries because of a previous failed merge. By default, if you try to check out such an entry from the index, the checkout operation will fail and nothing will be checked out. Using *-f* will ignore these unmerged entries. The contents from a specific side of the merge can be checked out of the index by using *--ours* or *--theirs*. With *-m*, changes made to the working tree file can be discarded to re-create the original conflicted merge result.

git checkout (-p|--patch) [<tree-ish>] [--] [<pathspec>…]

This is similar to the previous mode, but lets you use the interactive interface to show the "diff" output and choose which hunks to use in the result. See below for the description of *--patch* option.

OPTIONS

-q , --quiet

Quiet, suppress feedback messages.

--progress , --no-progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless *--quiet* is specified. This flag enables progress reporting even if not attached to a terminal, regardless of *--quiet*.

-f , --force

When switching branches, proceed even if the index or the working tree differs from *HEAD*, and even if there are untracked files in the way. This is used to throw away local changes and any untracked files or directories that are in the way.

When checking out paths from the index, do not fail upon unmerged entries; instead, unmerged entries are ignored.

--ours , --theirs

When checking out paths from the index, check out stage #2 (*ours*) or #3 (*theirs*) for unmerged paths.

Note that during *git rebase* and *git pull --rebase*, *ours* and *theirs* may appear swapped; *--ours* gives the version from the branch the changes are rebased onto, while *--theirs* gives the version from the branch that holds your work that is being rebased.

This is because *rebase* is used in a workflow that treats the history at the remote as the shared canonical one, and treats the work done on the branch you are rebasing as the third-party work to be integrated, and you are temporarily assuming the role of the keeper of the canonical history during the rebase. As the keeper of the canonical history, you need to view the history from the remote as *ours* (i.e. "our shared canonical history"), while what you did on your side branch as *theirs* (i.e. "one contributor's work on top of it").

-b <new-branch>

Create a new branch named <*new-branch*>, start it at <*start-point*>, and check the resulting branch out; see Section G.3.10, “git-branch(1)” for details.

-B <new-branch>

Creates the branch <*new-branch*>, start it at <*start-point*>; if it already exists, then reset it to <*start-point*>. And then check the resulting branch out. This is equivalent to running "git branch" with "-f" followed by "git checkout" of that branch; see Section G.3.10, “git-branch(1)” for details.

-t , --track[=(direct|inherit)]

When creating a new branch, set up "upstream" configuration. See "--track" in Section G.3.10, “git-branch(1)” for details.

If no *-b* option is given, the name of the new branch will be derived from the remote-tracking branch, by looking at the local part of the refspec configured for the corresponding remote, and then stripping the initial part up to the "*". This would tell us to use *hack* as the local branch when branching off of *origin/hack* (or *remotes/origin/hack*, or even *refs/remotes/origin/hack*). If the given name has no slash, or the above guessing results in an empty name, the guessing is aborted. You can explicitly give a name with *-b* in such a case.

--no-track

Do not set up "upstream" configuration, even if the *branch.autoSetupMerge* configuration variable is true.

--guess , --no-guess

If <*branch*> is not found but there does exist a tracking branch in exactly one remote (call it <*remote*>) with a matching name, treat as equivalent to

```
$ git checkout -b <branch> --track <remote>/<branch>
```

If the branch exists in multiple remotes and one of them is named by the *checkout.defaultRemote* configuration variable, we'll use that one for the purposes of disambiguation, even if the <*branch*> isn't unique across all remotes. Set it to e.g. *checkout.defaultRemote=origin* to always checkout remote branches from there if <*branch*> is ambiguous but exists on the *origin* remote. See also *checkout.defaultRemote* in Section G.3.29, “git-config(1)”.

--guess is the default behavior. Use *--no-guess* to disable it.

The default behavior can be set via the *checkout.guess* configuration variable.

-l

Create the new branch's reflog; see Section G.3.10, “git-branch(1)” for details.

-d , --detach

Rather than checking out a branch to work on it, check out a commit for inspection and discardable experiments. This is the default behavior of *git checkout* <*commit*> when <*commit*> is not a branch name. See the "DETACHED HEAD" section below for details.

--orphan <new-branch>

Create a new unborn branch, named *<new-branch>*, started from *<start-point>* and switch to it. The first commit made on this new branch will have no parents and it will be the root of a new history totally disconnected from all the other branches and commits.

The index and the working tree are adjusted as if you had previously run *git checkout <start-point>*. This allows you to start a new history that records a set of paths similar to *<start-point>* by easily running *git commit -a* to make the root commit.

This can be useful when you want to publish the tree from a commit without exposing its full history. You might want to do this to publish an open source branch of a project whose current tree is "clean", but whose full history contains proprietary or otherwise encumbered bits of code.

If you want to start a disconnected history that records a set of paths that is totally different from the one of *<start-point>*, then you should clear the index and the working tree right after creating the orphan branch by running *git rm -rf .* from the top level of the working tree. Afterwards you will be ready to prepare your new files, repopulating the working tree, by copying them from elsewhere, extracting a tarball, etc.

--ignore-skip-worktree-bits

In sparse checkout mode, *git checkout -- <paths>* would update only entries matched by *<paths>* and sparse patterns in *$GIT_DIR/info/sparse-checkout*. This option ignores the sparse patterns and adds back any files in *<paths>*.

-m , --merge

When switching branches, if you have local modifications to one or more files that are different between the current branch and the branch to which you are switching, the command refuses to switch branches in order to preserve your modifications in context. However, with this option, a three-way merge between the current branch, your working tree contents, and the new branch is done, and you will be on the new branch.

When a merge conflict happens, the index entries for conflicting paths are left unmerged, and you need to resolve the conflicts and mark the resolved paths with *git add* (or *git rm* if the merge should result in deletion of the path).

When checking out paths from the index, this option lets you recreate the conflicted merge in the specified paths. This option cannot be used when checking out paths from a tree-ish.

When switching branches with *--merge*, staged changes may be lost.

--conflict=<style>

The same as *--merge* option above, but changes the way the conflicting hunks are presented, overriding the *merge.conflictStyle* configuration variable. Possible values are "merge" (default), "diff3", and "zdiff3".

-p , --patch

Interactively select hunks in the difference between the *<tree-ish>* (or the index, if unspecified) and the working tree. The chosen hunks are then applied in reverse to the working tree (and if a *<tree-ish>* was specified, the index).

This means that you can use *git checkout -p* to selectively discard edits from your current working tree. See the Interactive Mode section of Section G.3.2, “git-add(1)” to learn how to operate the *--patch* mode.

Note that this option uses the no overlay mode by default (see also *--overlay*), and currently doesn't support overlay mode.

--ignore-other-worktrees

git checkout refuses when the wanted ref is already checked out by another worktree. This option makes it check the ref out anyway. In other words, the ref can be held by more than one worktree.

--overwrite-ignore , --no-overwrite-ignore

Silently overwrite ignored files when switching branches. This is the default behavior. Use *--no-overwrite-ignore* to abort the operation when the new branch contains ignored files.

--recurse-submodules , --no-recurse-submodules

Using *--recurse-submodules* will update the content of all active submodules according to the commit recorded in the superproject. If local modifications in a submodule would be overwritten the checkout will fail unless *-f* is used. If nothing (or *--no-recurse-submodules*) is used, submodules working trees will not be updated. Just like Section G.3.141, "git-submodule(1)", this will detach *HEAD* of the submodule.

--overlay , --no-overlay

In the default overlay mode, *git checkout* never removes files from the index or the working tree. When specifying *--no-overlay*, files that appear in the index and working tree, but not in <*tree-ish*> are removed, to make them match <*tree-ish*> exactly.

--pathspec-from-file=<file>

Pathspec is passed in <*file*> instead of commandline args. If <*file*> is exactly - then standard input is used. Pathspec elements are separated by LF or CR/LF. Pathspec elements can be quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, "git-config(1)"). See also *--pathspec-file-nul* and global *--literal-pathspecs*.

--pathspec-file-nul

Only meaningful with *--pathspec-from-file*. Pathspec elements are separated with NUL character and all other characters are taken literally (including newlines and quotes).

<branch>

Branch to checkout; if it refers to a branch (i.e., a name that, when prepended with "refs/heads/", is a valid ref), then that branch is checked out. Otherwise, if it refers to a valid commit, your *HEAD* becomes "detached" and you are no longer on any branch (see below for details).

You can use the *@{-N}* syntax to refer to the N-th last branch/commit checked out using "git checkout" operation. You may also specify - which is synonymous to *@{-1}*.

As a special case, you may use *A...B* as a shortcut for the merge base of *A* and *B* if there is exactly one merge base. You can leave out at most one of *A* and *B*, in which case it defaults to *HEAD*.

<new-branch>

Name for the new branch.

<start-point>

The name of a commit at which to start the new branch; see Section G.3.10, "git-branch(1)" for details. Defaults to *HEAD*.

As a special case, you may use *"A...B"* as a shortcut for the merge base of *A* and *B* if there is exactly one merge base. You can leave out at most one of *A* and *B*, in which case it defaults to *HEAD*.

<tree-ish>

Tree to checkout from (when paths are given). If not specified, the index will be used.

As a special case, you may use *"A...B"* as a shortcut for the merge base of *A* and *B* if there is exactly one merge base. You can leave out at most one of *A* and *B*, in which case it defaults to *HEAD*.

--

Do not interpret any more arguments as options.

<pathspec>…

Limits the paths affected by the operation.

For more details, see the *pathspec* entry in Section G.4.19, “gitglossary(7)”.

DETACHED HEAD

HEAD normally refers to a named branch (e.g. *master*). Meanwhile, each branch refers to a specific commit. Let's look at a repo with three commits, one of them tagged, and with branch *master* checked out:

```
           HEAD (refers to branch 'master')
            |
            v
a---b---c  branch 'master' (refers to commit 'c')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

When a commit is created in this state, the branch is updated to refer to the new commit. Specifically, *git commit* creates a new commit *d*, whose parent is commit *c*, and then updates branch *master* to refer to new commit *d*. *HEAD* still refers to branch *master* and so indirectly now refers to commit *d*:

```
$ edit; git add; git commit

               HEAD (refers to branch 'master')
                |
                v
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

It is sometimes useful to be able to checkout a commit that is not at the tip of any named branch, or even to create a new commit that is not referenced by a named branch. Let's look at what happens when we checkout commit *b* (here we show two ways this may be done):

```
$ git checkout v2.0  # or
$ git checkout master^^

   HEAD (refers to commit 'b')
    |
    v
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

Notice that regardless of which checkout command we use, *HEAD* now refers directly to commit *b*. This is known as being in detached *HEAD* state. It means simply that *HEAD* refers to a specific commit, as opposed to referring to a named branch. Let's see what happens when we create a commit:

```
$ edit; git add; git commit

     HEAD (refers to commit 'e')
      |
      v
      e
     /
a---b---c---d  branch 'master' (refers to commit 'd')
```

```
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

There is now a new commit *e*, but it is referenced only by *HEAD*. We can of course add yet another commit in this state:

```
$ edit; git add; git commit

         HEAD (refers to commit 'f')
          |
          v
      e---f
     /
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

In fact, we can perform all the normal Git operations. But, let's look at what happens when we then checkout *master*:

```
$ git checkout master

               HEAD (refers to branch 'master')
      e---f     |
     /          v
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

It is important to realize that at this point nothing refers to commit *f*. Eventually commit *f* (and by extension commit *e*) will be deleted by the routine Git garbage collection process, unless we create a reference before that happens. If we have not yet moved away from commit *f*, any of these will create a reference to it:

```
$ git checkout -b foo  # or "git switch -c foo"  ❶
$ git branch foo                                 ❷
$ git tag foo                                    ❸
```

❶ creates a new branch *foo*, which refers to commit *f*, and then updates *HEAD* to refer to branch *foo*. In other words, we'll no longer be in detached *HEAD* state after this command.
❷ similarly creates a new branch *foo*, which refers to commit *f*, but leaves *HEAD* detached.
❸ creates a new tag *foo*, which refers to commit *f*, leaving *HEAD* detached.

If we have moved away from commit *f*, then we must first recover its object name (typically by using git reflog), and then we can create a reference to it. For example, to see the last two commits to which *HEAD* referred, we can use either of these commands:

```
$ git reflog -2 HEAD # or
$ git log -g -2 HEAD
```

ARGUMENT DISAMBIGUATION

When there is only one argument given and it is not -- (e.g. *git checkout abc*), and when the argument is both a valid <*tree-ish*> (e.g. a branch *abc* exists) and a valid <*pathspec*> (e.g. a file or a directory whose name is "abc" exists), Git would usually ask you to disambiguate. Because checking out a branch is so common an operation, however, *git checkout abc* takes "abc" as a <*tree-ish*> in such a situation. Use *git checkout* -- <*pathspec*> if you want to checkout these paths out of the index.

EXAMPLES

1. 1. Paths

The following sequence checks out the *master* branch, reverts the *Makefile* to two revisions back, deletes *hello.c* by mistake, and gets it back from the index.

```
$ git checkout master             ❶
$ git checkout master~2 Makefile  ❷
$ rm -f hello.c
$ git checkout hello.c            ❸
```

❶ switch branch
❷ take a file out of another commit
❸ restore *hello.c* from the index

If you want to check out *all* C source files out of the index, you can say

```
$ git checkout -- '*.c'
```

Note the quotes around **.c*. The file *hello.c* will also be checked out, even though it is no longer in the working tree, because the file globbing is used to match entries in the index (not in the working tree by the shell).

If you have an unfortunate branch that is named *hello.c*, this step would be confused as an instruction to switch to that branch. You should instead write:

```
$ git checkout -- hello.c
```

2. 2. Merge

After working in the wrong branch, switching to the correct branch would be done using:

```
$ git checkout mytopic
```

However, your "wrong" branch and correct *mytopic* branch may differ in files that you have modified locally, in which case the above checkout would fail like this:

```
$ git checkout mytopic
error: You have local changes to 'frotz'; not switching branches.
```

You can give the *-m* flag to the command, which would try a three-way merge:

```
$ git checkout -m mytopic
Auto-merging frotz
```

After this three-way merge, the local modifications are *not* registered in your index file, so *git diff* would show you what changes you made since the tip of the new branch.

3. 3. Merge conflict

When a merge conflict happens during switching branches with the *-m* option, you would see something like this:

```
$ git checkout -m mytopic
Auto-merging frotz
ERROR: Merge conflict in frotz
fatal: merge program failed
```

At this point, *git diff* shows the changes cleanly merged as in the previous example, as well as the changes in the conflicted files. Edit and resolve the conflict and mark it resolved with *git add* as usual:

```
$ edit frotz
```

```
$ git add frotz
```

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

checkout.defaultRemote

When you run *git checkout* <*something*> or *git switch* <*something*> and only have one remote, it may implicitly fall back on checking out and tracking e.g. *origin/*<*something*>. This stops working as soon as you have more than one remote with a <*something*> reference. This setting allows for setting the name of a preferred remote that should always win when it comes to disambiguation. The typical use-case is to set this to *origin*.

Currently this is used by Section G.3.140, “git-switch(1)” and Section G.3.19, “git-checkout(1)” when *git checkout* <*something*> or *git switch* <*something*> will checkout the <*something*> branch on another remote, and by Section G.3.159, “git-worktree(1)” when *git worktree add* refers to a remote branch. This setting might be used for other checkout-like commands or functionality in the future.

checkout.guess

Provides the default value for the *--guess* or *--no-guess* option in *git checkout* and *git switch*. See Section G.3.140, “git-switch(1)” and Section G.3.19, “git-checkout(1)”.

checkout.workers

The number of parallel workers to use when updating the working tree. The default is one, i.e. sequential execution. If set to a value less than one, Git will use as many workers as the number of logical cores available. This setting and *checkout.thresholdForParallelism* affect all commands that perform checkout. E.g. checkout, clone, reset, sparse-checkout, etc.

Note: Parallel checkout usually delivers better performance for repositories located on SSDs or over NFS. For repositories on spinning disks and/or machines with a small number of cores, the default sequential checkout often performs better. The size and compression level of a repository might also influence how well the parallel version performs.

checkout.thresholdForParallelism

When running parallel checkout with a small number of files, the cost of subprocess spawning and inter-process communication might outweigh the parallelization gains. This setting allows you to define the minimum number of files for which parallel checkout should be attempted. The default is 100.

SEE ALSO

Section G.3.140, “git-switch(1)”, Section G.3.119, “git-restore(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.20. git-cherry-pick(1)

2024-07-02

Revision History

2024-07-02

NAME

git-cherry-pick - Apply the changes introduced by some existing commits

SYNOPSIS

git cherry-pick [--edit] [-n] [-m <parent-number>] [-s] [-x] [--ff]

[-S[<keyid>]] <commit>…
git cherry-pick (--continue | --skip | --abort | --quit)

DESCRIPTION

Given one or more existing commits, apply the change each one introduces, recording a new commit for each. This requires your working tree to be clean (no modifications from the HEAD commit).

When it is not obvious how to apply a change, the following happens:

1. The current branch and *HEAD* pointer stay at the last commit successfully made.
2. The *CHERRY_PICK_HEAD* ref is set to point at the commit that introduced the change that is difficult to apply.
3. Paths in which the change applied cleanly are updated both in the index file and in your working tree.
4. For conflicting paths, the index file records up to three versions, as described in the "TRUE MERGE" section of Section G.3.86, "git-merge(1)". The working tree files will include a description of the conflict bracketed by the usual conflict markers <<<<<<< and >>>>>>>.
5. No other modifications are made.

See Section G.3.86, "git-merge(1)" for some hints on resolving such conflicts.

OPTIONS

<commit>…

Commits to cherry-pick. For a more complete list of ways to spell commits, see Section G.4.14, "gitrevisions(7)". Sets of commits can be passed but no traversal is done by default, as if the *--no-walk* option was specified, see Section G.3.120, "git-rev-list(1)". Note that specifying a range will feed all <commit>… arguments to a single revision walk (see a later example that uses *maint master..next*).

-e , --edit

With this option, *git cherry-pick* will let you edit the commit message prior to committing.

--cleanup=<mode>

This option determines how the commit message will be cleaned up before being passed on to the commit machinery. See Section G.3.28, "git-commit(1)" for more details. In particular, if the <*mode*> is given a value of *scissors*, scissors will be appended to *MERGE_MSG* before being passed on in the case of a conflict.

-x

When recording the commit, append a line that says "(cherry picked from commit …)" to the original commit message in order to indicate which commit this change was cherry-picked from. This is done only for cherry picks without conflicts. Do not use this option if you are cherry-picking from your private branch because the information is useless to the recipient. If on the other hand you are cherry-picking between two publicly visible branches (e.g. backporting a fix to a maintenance branch for an older release from a development branch), adding this information can be useful.

-r

It used to be that the command defaulted to do *-x* described above, and *-r* was to disable it. Now the default is not to do *-x* so this option is a no-op.

-m <parent-number> , --mainline <parent-number>

Usually you cannot cherry-pick a merge because you do not know which side of the merge should be considered the mainline. This option specifies the parent number (starting from 1) of the mainline and allows cherry-pick to replay the change relative to the specified parent.

-n , --no-commit

Usually the command automatically creates a sequence of commits. This flag applies the changes necessary to cherry-pick each named commit to your working tree and the index, without making any commit. In addition, when this option is used, your index does not have to match the HEAD commit. The cherry-pick is done against the beginning state of your index.

This is useful when cherry-picking more than one commits' effect to your index in a row.

-s , --signoff

Add a *Signed-off-by* trailer at the end of the commit message. See the signoff option in Section G.3.28, “git-commit(1)” for more information.

-S[<keyid>] , --gpg-sign[=<keyid>] , --no-gpg-sign

GPG-sign commits. The *keyid* argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space. *--no-gpg-sign* is useful to countermand both *commit.gpgSign* configuration variable, and earlier *--gpg-sign*.

--ff

If the current HEAD is the same as the parent of the cherry-pick'ed commit, then a fast forward to this commit will be performed.

--allow-empty

By default, cherry-picking an empty commit will fail, indicating that an explicit invocation of *git commit --allow-empty* is required. This option overrides that behavior, allowing empty commits to be preserved automatically in a cherry-pick. Note that when "--ff" is in effect, empty commits that meet the "fast-forward" requirement will be kept even without this option. Note also, that use of this option only keeps commits that were initially empty (i.e. the commit recorded the same tree as its parent). Commits which are made empty due to a previous commit will cause the cherry-pick to fail. To force the inclusion of those commits, use *--empty=keep*.

--allow-empty-message

By default, cherry-picking a commit with an empty message will fail. This option overrides that behavior, allowing commits with empty messages to be cherry picked.

--empty=(drop|keep|stop)

How to handle commits being cherry-picked that are redundant with changes already in the current history.

drop

The commit will be dropped.

keep

The commit will be kept. Implies *--allow-empty*.

stop

The cherry-pick will stop when the commit is applied, allowing you to examine the commit. This is the default behavior.

Note that *--empty=drop* and *--empty=stop* only specify how to handle a commit that was not initially empty, but rather became empty due to a previous commit. Commits that were initially empty will still cause the cherry-pick to fail unless one of *--empty=keep* or *--allow-empty* are specified.

--keep-redundant-commits

Deprecated synonym for *--empty=keep*.

--strategy=<strategy>

Use the given merge strategy. Should only be used once. See the MERGE STRATEGIES section in Section G.3.86, “git-merge(1)” for details.

-X<option> , --strategy-option=<option>

Pass the merge strategy-specific option through to the merge strategy. See Section G.3.86, “git-merge(1)” for details.

--rerere-autoupdate , --no-rerere-autoupdate

After the rerere mechanism reuses a recorded resolution on the current conflict to update the files in the working tree, allow it to also update the index with the result of resolution. *--no-rerere-autoupdate* is a good way to double-check what *rerere* did and catch potential mismerges, before committing the result to the index with a separate *git add*.

SEQUENCER SUBCOMMANDS

--continue

Continue the operation in progress using the information in *.git/sequencer*. Can be used to continue after resolving conflicts in a failed cherry-pick or revert.

--skip

Skip the current commit and continue with the rest of the sequence.

--quit

Forget about the current operation in progress. Can be used to clear the sequencer state after a failed cherry-pick or revert.

--abort

Cancel the operation and return to the pre-sequence state.

EXAMPLES

git cherry-pick master

Apply the change introduced by the commit at the tip of the master branch and create a new commit with this change.

git cherry-pick ..master , *git cherry-pick ^HEAD master*

Apply the changes introduced by all commits that are ancestors of master but not of HEAD to produce new commits.

git cherry-pick maint next ^master , *git cherry-pick maint master..next*

Apply the changes introduced by all commits that are ancestors of maint or next, but not master or any of its ancestors. Note that the latter does not mean *maint* and everything between *master* and *next*; specifically, *maint* will not be used if it is included in *master*.

git cherry-pick master~4 master~2

Apply the changes introduced by the fifth and third last commits pointed to by master and create 2 new commits with these changes.

git cherry-pick -n master~1 next

Apply to the working tree and the index the changes introduced by the second last commit pointed to by master and by the last commit pointed to by next, but do not create any commit with these changes.

git cherry-pick --ff ..next

If history is linear and HEAD is an ancestor of next, update the working tree and advance the HEAD pointer to match next. Otherwise, apply the changes introduced by those commits that are in next but not HEAD to the current branch, creating a new commit for each new change.

git rev-list --reverse master -- README | git cherry-pick -n --stdin

Apply the changes introduced by all commits on the master branch that touched README to the working tree and index, so the result can be inspected and made into a single new commit if suitable.

The following sequence attempts to backport a patch, bails out because the code the patch applies to has changed too much, and then tries again, this time exercising more care about matching up context lines.

```
$ git cherry-pick topic^             ❶
$ git diff                           ❷
$ git cherry-pick --abort            ❸
$ git cherry-pick -Xpatience topic^  ❹
```

❶ apply the change that would be shown by *git show topic^*. In this example, the patch does not apply cleanly, so information about the conflict is written to the index and working tree and no new commit results.
❷ summarize changes to be reconciled
❸ cancel the cherry-pick. In other words, return to the pre-cherry-pick state, preserving any local modifications you had in the working tree.
❹ try to apply the change introduced by *topic^* again, spending extra time to avoid mistakes based on incorrectly matching context lines.

SEE ALSO

Section G.3.122, “git-revert(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.21. git-cherry(1)

2024-07-02
Revision History
2024-07-02

NAME

git-cherry - Find commits yet to be applied to upstream

SYNOPSIS

git cherry [-v] [<upstream> [<head> [<limit>]]]

DESCRIPTION

Determine whether there are commits in *<head>..<upstream>* that are equivalent to those in the range *<limit>..<head>*.

The equivalence test is based on the diff, after removing whitespace and line numbers. git-cherry therefore detects when commits have been "copied" by means of Section G.3.20, “git-cherry-pick(1)”, Section G.3.3, “git-am(1)” or Section G.3.107, “git-rebase(1)”.

Outputs the SHA1 of every commit in *<limit>..<head>*, prefixed with - for commits that have an equivalent in <upstream>, and + for commits that do not.

OPTIONS

-v

Show the commit subjects next to the SHA1s.

<upstream>

Upstream branch to search for equivalent commits. Defaults to the upstream branch of HEAD.

<head>

Working branch; defaults to HEAD.

<limit>

Do not report commits up to (and including) limit.

EXAMPLES

1. Patch workflows

git-cherry is frequently used in patch-based workflows (see Section G.4.18, “gitworkflows(7)”) to determine if a series of patches has been applied by the upstream maintainer. In such a workflow you might create and send a topic branch like this:

```
$ git checkout -b topic origin/master
# work and create some commits
$ git format-patch origin/master
$ git send-email ... 00*
```

Later, you can see whether your changes have been applied by saying (still on *topic*):

```
$ git fetch  # update your notion of origin/master
$ git cherry -v
```

2. Concrete example

In a situation where topic consisted of three commits, and the maintainer applied two of them, the situation might look like:

```
$ git log --graph --oneline --decorate --boundary origin/master...topic
* 7654321 (origin/master) upstream tip commit
[... snip some other commits ...]
* cccc111 cherry-pick of C
* aaaa111 cherry-pick of A
[... snip a lot more that has happened ...]
| * cccc000 (topic) commit C
| * bbbb000 commit B
| * aaaa000 commit A
|/
o 1234567 branch point
```

In such cases, git-cherry shows a concise summary of what has yet to be applied:

```
$ git cherry origin/master topic
- cccc000... commit C
+ bbbb000... commit B
```

```
- aaaa000... commit A
```

Here, we see that the commits A and C (marked with -) can be dropped from your *topic* branch when you rebase it on top of *origin/master*, while the commit B (marked with +) still needs to be kept so that it will be sent to be applied to *origin/master*.

3. Using a limit

The optional <limit> is useful in cases where your topic is based on other work that is not in upstream. Expanding on the previous example, this might look like:

```
$ git log --graph --oneline --decorate --boundary origin/master...topic
* 7654321 (origin/master) upstream tip commit
[... snip some other commits ...]
* cccc111 cherry-pick of C
* aaaa111 cherry-pick of A
[... snip a lot more that has happened ...]
| * cccc000 (topic) commit C
| * bbbb000 commit B
| * aaaa000 commit A
| * 0000fff (base) unpublished stuff F
[... snip ...]
| * 0000aaa unpublished stuff A
|/
o 1234567 merge-base between upstream and topic
```

By specifying *base* as the limit, you can avoid listing commits between *base* and *topic*:

```
$ git cherry origin/master topic base
- cccc000... commit C
+ bbbb000... commit B
- aaaa000... commit A
```

SEE ALSO

Section G.3.99, “git-patch-id(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.22. git-citool(1)

2024-07-02

Revision History

2024-07-02

NAME

git-citool - Graphical alternative to git-commit

SYNOPSIS

git citool

DESCRIPTION

A Tcl/Tk based graphical interface to review modified files, stage them into the index, enter a commit message and record the new commit onto the current branch. This interface is an alternative to the less interactive *git commit* program.

git citool is actually a standard alias for *git gui citool*. See Section G.3.61, “git-gui(1)” for more details.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.23. git-clean(1)

2024-07-02

Revision History

2024-07-02

NAME

git-clean - Remove untracked files from the working tree

SYNOPSIS

git clean [-d] [-f] [-i] [-n] [-q] [-e <pattern>] [-x | -X] [--] [<pathspec>…]

DESCRIPTION

Cleans the working tree by recursively removing files that are not under version control, starting from the current directory.

Normally, only files unknown to Git are removed, but if the *-x* option is specified, ignored files are also removed. This can, for example, be useful to remove all build products.

If any optional *<pathspec>*... arguments are given, only those paths that match the pathspec are affected.

OPTIONS

-d

Normally, when no <pathspec> is specified, git clean will not recurse into untracked directories to avoid removing too much. Specify -d to have it recurse into such directories as well. If a <pathspec> is specified, -d is irrelevant; all untracked files matching the specified paths (with exceptions for nested git directories mentioned under *--force*) will be removed.

-f , --force

If the Git configuration variable clean.requireForce is not set to false, *git clean* will refuse to delete files or directories unless given -f. Git will refuse to modify untracked nested git repositories (directories with a .git subdirectory) unless a second -f is given.

-i , --interactive

Show what would be done and clean files interactively. See Interactive mode for details. Configuration variable *clean.requireForce* is ignored, as this mode gives its own safety protection by going interactive.

-n , --dry-run

Don't actually remove anything, just show what would be done. Configuration variable *clean.requireForce* is ignored, as nothing will be deleted anyway.

-q , --quiet

Be quiet, only report errors, but not the files that are successfully removed.

-e <pattern> , --exclude=<pattern>

Use the given exclude pattern in addition to the standard ignore rules (see Section G.4.5, “gitignore(5)”).

-x

Don't use the standard ignore rules (see Section G.4.5, “gitignore(5)”), but still use the ignore rules given with *-e* options from the command line. This allows removing all untracked files, including build products.

This can be used (possibly in conjunction with *git restore* or *git reset*) to create a pristine working directory to test a clean build.

-X

Remove only files ignored by Git. This may be useful to rebuild everything from scratch, but keep manually created files.

Interactive mode

When the command enters the interactive mode, it shows the files and directories to be cleaned, and goes into its interactive command loop.

The command loop shows the list of subcommands available, and gives a prompt "What now> ". In general, when the prompt ends with a single >, you can pick only one of the choices given and type return, like this:

```
    *** Commands ***
        1: clean                2: filter by pattern    3: select by
 numbers
        4: ask each             5: quit                 6: help
    What now> 1
```

You also could say *c* or *clean* above as long as the choice is unique.

The main command loop has 6 subcommands.

clean

Start cleaning files and directories, and then quit.

filter by pattern

This shows the files and directories to be deleted and issues an "Input ignore patterns>>" prompt. You can input space-separated patterns to exclude files and directories from deletion. E.g. "*.c *.h" will exclude files ending with ".c" and ".h" from deletion. When you are satisfied with the filtered result, press ENTER (empty) back to the main menu.

select by numbers

This shows the files and directories to be deleted and issues an "Select items to delete>>" prompt. When the prompt ends with double >> like this, you can make more than one selection, concatenated with whitespace or comma. Also you can say ranges. E.g. "2-5 7,9" to choose 2,3,4,5,7,9 from the list. If the second number in a range is omitted, all remaining items are selected. E.g. "7-" to choose 7,8,9 from the list. You can say * to choose everything. Also when you are satisfied with the filtered result, press ENTER (empty) back to the main menu.

ask each

This will start to clean, and you must confirm one by one in order to delete items. Please note that this action is not as efficient as the above two actions.

quit

This lets you quit without doing any cleaning.

help

Show brief usage of interactive git-clean.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

clean.requireForce

A boolean to make git-clean refuse to delete files unless -f is given. Defaults to true.

SEE ALSO

Section G.4.5, “gitignore(5)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.24. git-clone(1)

2024-07-02
Revision History
2024-07-02

NAME

git-clone - Clone a repository into a new directory

SYNOPSIS

git clone [`--template=`*<template-directory>*]
[*-l*] [*-s*] [*--no-hardlinks*] [*-q*] [*-n*] [*--bare*] [*--mirror*]
[*-o <name>*] [*-b <name>*] [*-u <upload-pack>*] [*--reference <repository>*]
[*--dissociate*] [*--separate-git-dir <git-dir>*]
[*--depth <depth>*] [*--*[*no-*]*single-branch*] [*--no-tags*]
[`--recurse-submodules`[*=<pathspec>*]] [*--*[*no-*]*shallow-submodules*]
[*--*[*no-*]*remote-submodules*] [*--jobs <n>*] [*--sparse*] [*--*[*no-*]*reject-shallow*]
[`--filter=`*<filter-spec>*] [*--also-filter-submodules*]] [*--*] *<repository>*
[*<directory>*]

DESCRIPTION

Clones a repository into a newly created directory, creates remote-tracking branches for each branch in the cloned repository (visible using *git branch --remotes*), and creates and checks out an initial branch that is forked from the cloned repository's currently active branch.

After the clone, a plain *git fetch* without arguments will update all the remote-tracking branches, and a *git pull* without arguments will in addition merge the remote master branch into the current master branch, if any (this is untrue when *--single-branch* is given; see below).

This default configuration is achieved by creating references to the remote branch heads under *refs/remotes/origin* and by initializing *remote.origin.url* and *remote.origin.fetch* configuration variables.

OPTIONS

-l , --local

When the repository to clone from is on a local machine, this flag bypasses the normal "Git aware" transport mechanism and clones the repository by making a copy of *HEAD* and everything under objects and refs directories. The files under *.git/objects/* directory are hardlinked to save space when possible.

If the repository is specified as a local path (e.g., */path/to/repo*), this is the default, and --local is essentially a no-op. If the repository is specified as a URL, then this flag is ignored (and we never use the local optimizations). Specifying *--no-local* will override the default when */path/to/repo* is given, using the regular Git transport instead.

If the repository's *$GIT_DIR/objects* has symbolic links or is a symbolic link, the clone will fail. This is a security measure to prevent the unintentional copying of files by dereferencing the symbolic links.

NOTE: this operation can race with concurrent modification to the source repository, similar to running *cp -r src dst* while modifying *src*.

--no-hardlinks

Force the cloning process from a repository on a local filesystem to copy the files under the *.git/objects* directory instead of using hardlinks. This may be desirable if you are trying to make a back-up of your repository.

-s , *--shared*

When the repository to clone is on the local machine, instead of using hard links, automatically setup *.git/objects/info/alternates* to share the objects with the source repository. The resulting repository starts out without any object of its own.

NOTE: this is a possibly dangerous operation; do **not** use it unless you understand what it does. If you clone your repository using this option and then delete branches (or use any other Git command that makes any existing commit unreferenced) in the source repository, some objects may become unreferenced (or dangling). These objects may be removed by normal Git operations (such as *git commit*) which automatically call *git maintenance run --auto*. (See Section G.3.80, "git-maintenance(1)".) If these objects are removed and were referenced by the cloned repository, then the cloned repository will become corrupt.

Note that running *git repack* without the *--local* option in a repository cloned with *--shared* will copy objects from the source repository into a pack in the cloned repository, removing the disk space savings of *clone --shared*. It is safe, however, to run *git gc*, which uses the *--local* option by default.

If you want to break the dependency of a repository cloned with *--shared* on its source repository, you can simply run *git repack -a* to copy all objects from the source repository into a pack in the cloned repository.

--reference[*-if-able*] *<repository>*

If the reference *<repository>* is on the local machine, automatically setup *.git/objects/info/alternates* to obtain objects from the reference *<repository>*. Using an already existing repository as an alternate will require fewer objects to be copied from the repository being cloned, reducing network and local storage costs. When using the *--reference-if-able*, a non existing directory is skipped with a warning instead of aborting the clone.

NOTE: see the NOTE for the *--shared* option, and also the *--dissociate* option.

--dissociate

Borrow the objects from reference repositories specified with the *--reference* options only to reduce network transfer, and stop borrowing from them after a clone is made by making necessary local copies of borrowed objects. This option can also be used when cloning locally from a repository that already borrows objects from another repository--the new repository will borrow objects from the same repository, and this option can be used to stop the borrowing.

-q , *--quiet*

Operate quietly. Progress is not reported to the standard error stream.

-v , *--verbose*

Run verbosely. Does not affect the reporting of progress status to the standard error stream.

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless *--quiet* is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

`--server-option=`*<option>*

Transmit the given string to the server when communicating using protocol version 2. The given string must not contain a NUL or LF character. The server's handling of server options, including unknown ones, is

server-specific. When multiple `--server-option=<option>` are given, they are all sent to the other side in the order listed on the command line.

-n , *--no-checkout*

No checkout of HEAD is performed after the clone is complete.

--[*no-*]*reject-shallow*

Fail if the source repository is a shallow repository. The *clone.rejectShallow* configuration variable can be used to specify the default.

--bare

Make a *bare* Git repository. That is, instead of creating *<directory>* and placing the administrative files in *<directory>/.git*, make the *<directory>* itself the *$GIT_DIR*. This obviously implies the *--no-checkout* because there is nowhere to check out the working tree. Also the branch heads at the remote are copied directly to corresponding local branch heads, without mapping them to *refs/remotes/origin/*. When this option is used, neither remote-tracking branches nor the related configuration variables are created.

--sparse

Employ a sparse-checkout, with only files in the toplevel directory initially being present. The Section G.3.135, "git-sparse-checkout(1)" command can be used to grow the working directory as needed.

`--filter=`*<filter-spec>*

Use the partial clone feature and request that the server sends a subset of reachable objects according to a given object filter. When using *--filter*, the supplied *<filter-spec>* is used for the partial clone filter. For example, *--filter=blob:none* will filter out all blobs (file contents) until needed by Git. Also, `--filter=blob:limit=`*<size>* will filter out all blobs of size at least *<size>*. For more details on filter specifications, see the *--filter* option in Section G.3.120, "git-rev-list(1)".

--also-filter-submodules

Also apply the partial clone filter to any submodules in the repository. Requires *--filter* and *--recurse-submodules*. This can be turned on by default by setting the *clone.filterSubmodules* config option.

--mirror

Set up a mirror of the source repository. This implies *--bare*. Compared to *--bare*, *--mirror* not only maps local branches of the source to local branches of the target, it maps all refs (including remote-tracking branches, notes etc.) and sets up a refspec configuration such that all these refs are overwritten by a *git remote update* in the target repository.

-o <name> , *--origin <name>*

Instead of using the remote name *origin* to keep track of the upstream repository, use *<name>*. Overrides *clone.defaultRemoteName* from the config.

-b <name> , *--branch <name>*

Instead of pointing the newly created HEAD to the branch pointed to by the cloned repository's HEAD, point to *<name>* branch instead. In a non-bare repository, this is the branch that will be checked out. *--branch* can also take tags and detaches the HEAD at that commit in the resulting repository.

-u <upload-pack> , *--upload-pack <upload-pack>*

When given, and the repository to clone from is accessed via ssh, this specifies a non-default path for the command run on the other end.

`--template=`*<template-directory>*

Specify the directory from which templates will be used; (See the "TEMPLATE DIRECTORY" section of Section G.3.71, “git-init(1)”.)

-c <key>=<value> , *--config <key>=<value>*

Set a configuration variable in the newly-created repository; this takes effect immediately after the repository is initialized, but before the remote history is fetched or any files checked out. The *<key>* is in the same format as expected by Section G.3.29, “git-config(1)” (e.g., *core.eol=true*). If multiple values are given for the same key, each value will be written to the config file. This makes it safe, for example, to add additional fetch refspecs to the origin remote.

Due to limitations of the current implementation, some configuration variables do not take effect until after the initial fetch and checkout. Configuration variables known to not take effect are: `remote.`*<name>*`.mirror` and `remote.`*<name>*`.tagOpt`. Use the corresponding *--mirror* and *--no-tags* options instead.

--depth <depth>

Create a *shallow* clone with a history truncated to the specified number of commits. Implies *--single-branch* unless *--no-single-branch* is given to fetch the histories near the tips of all branches. If you want to clone submodules shallowly, also pass *--shallow-submodules*.

`--shallow-since=`*<date>*

Create a shallow clone with a history after the specified time.

`--shallow-exclude=`*<revision>*

Create a shallow clone with a history, excluding commits reachable from a specified remote branch or tag. This option can be specified multiple times.

--[*no-*]*single-branch*

Clone only the history leading to the tip of a single branch, either specified by the *--branch* option or the primary branch remote's *HEAD* points at. Further fetches into the resulting repository will only update the remote-tracking branch for the branch this option was used for the initial cloning. If the *HEAD* at the remote did not point at any branch when *--single-branch* clone was made, no remote-tracking branch is created.

--no-tags

Don't clone any tags, and set *remote.<remote>.tagOpt=--no-tags* in the config, ensuring that future *git pull* and *git fetch* operations won't follow any tags. Subsequent explicit tag fetches will still work, (see Section G.3.49, “git-fetch(1)”).

Can be used in conjunction with *--single-branch* to clone and maintain a branch with no references other than a single cloned branch. This is useful e.g. to maintain minimal clones of the default branch of some repository for search indexing.

--recurse-submodules[*=<pathspec>*]

After the clone is created, initialize and clone submodules within based on the provided *<pathspec>*. If no *=<pathspec>* is provided, all submodules are initialized and cloned. This option can be given multiple times for pathspecs consisting of multiple entries. The resulting clone has *submodule.active* set to the provided pathspec, or "." (meaning all submodules) if no pathspec is provided.

Submodules are initialized and cloned using their default settings. This is equivalent to running *git submodule update --init --recursive <pathspec>* immediately after the clone is finished. This option is ignored if the cloned repository does not have a worktree/checkout (i.e. if any of *--no-checkout*/*-n*, *--bare*, or *--mirror* is given)

--[*no*-]*shallow-submodules*

All submodules which are cloned will be shallow with a depth of 1.

--[*no*-]*remote-submodules*

All submodules which are cloned will use the status of the submodule's remote-tracking branch to update the submodule, rather than the superproject's recorded SHA-1. Equivalent to passing *--remote* to *git submodule update*.

--separate-git-dir=<git-dir>

Instead of placing the cloned repository where it is supposed to be, place the cloned repository at the specified directory, then make a filesystem-agnostic Git symbolic link to there. The result is Git repository can be separated from working tree.

--ref-format=<ref-format>

Specify the given ref storage format for the repository. The valid values are:

- *files* for loose files with packed-refs. This is the default.
- *reftable* for the reftable format. This format is experimental and its internals are subject to change.

-j <n> , *--jobs <n>*

The number of submodules fetched at the same time. Defaults to the *submodule.fetchJobs* option.

<repository>

The (possibly remote) *<repository>* to clone from. See the GIT URLS section below for more information on specifying repositories.

<directory>

The name of a new directory to clone into. The "humanish" part of the source repository is used if no *<directory>* is explicitly given (*repo* for */path/to/repo.git* and *foo* for *host.xz:foo/.git*). Cloning into an existing directory is only allowed if the directory is empty.

--bundle-uri=<uri>

Before fetching from the remote, fetch a bundle from the given *<uri>* and unbundle the data into the local repository. The refs in the bundle will be stored under the hidden *refs/bundle/** namespace. This option is incompatible with *--depth*, *--shallow-since*, and *--shallow-exclude*.

GIT URLS

In general, URLs contain information about the transport protocol, the address of the remote server, and the path to the repository. Depending on the transport protocol, some of this information may be absent.

Git supports ssh, git, http, and https protocols (in addition, ftp and ftps can be used for fetching, but this is inefficient and deprecated; do not use them).

The native transport (i.e. git:// URL) does no authentication and should be used with caution on unsecured networks.

The following syntaxes may be used with them:

- `ssh://`[*<user>*`@`]*<host>*[`:`*<port>*]`/`*<path-to-git-repo>*
- `git://`*<host>*[:*<port>*]`/`*<path-to-git-repo>*

- `http[s]://`*<host>*`[:`*<port>*`]/`*<path-to-git-repo>*
- `ftp[s]://`*<host>*`[:`*<port>*`]/`*<path-to-git-repo>*

An alternative scp-like syntax may also be used with the ssh protocol:

- [*<user>*`@`]*<host>*`:/`*<path-to-git-repo>*

This syntax is only recognized if there are no slashes before the first colon. This helps differentiate a local path that contains a colon. For example the local path *foo:bar* could be specified as an absolute path or *./foo:bar* to avoid being misinterpreted as an ssh url.

The ssh and git protocols additionally support `~`*<username>* expansion:

- `ssh://`[*<user>*`@`]*<host>*`[:`*<port>*`]/~`*<user>*`/`*<path-to-git-repo>*
- `git://`*<host>*`[:`*<port>*`]/~`*<user>*`/`*<path-to-git-repo>*
- [*<user>*`@`]*<host>*`:~`*<user>*`/`*<path-to-git-repo>*

For local repositories, also supported by Git natively, the following syntaxes may be used:

- */path/to/repo.git/*
- `file:///path/to/repo.git/`

These two syntaxes are mostly equivalent, except the former implies *--local* option.

git clone, *git fetch* and *git pull*, but not *git push*, will also accept a suitable bundle file. See Section G.3.12, “git-bundle(1)”.

When Git doesn't know how to handle a certain transport protocol, it attempts to use the *remote-<transport>* remote helper, if one exists. To explicitly request a remote helper, the following syntax may be used:

- *<transport>*::*<address>*

where *<address>* may be a path, a server and path, or an arbitrary URL-like string recognized by the specific remote helper being invoked. See Section G.4.12, “gitremote-helpers(7)” for details.

If there are a large number of similarly-named remote repositories and you want to use a different format for them (such that the URLs you use will be rewritten into URLs that work), you can create a configuration section of the form:

```
[url "<actual-url-base>"]
    insteadOf = <other-url-base>
```

For example, with this:

```
[url "git://git.host.xz/"]
        insteadOf = host.xz:/path/to/
        insteadOf = work:
```

a URL like "work:repo.git" or like "host.xz:/path/to/repo.git" will be rewritten in any context that takes a URL to be "git://git.host.xz/repo.git".

If you want to rewrite URLs for push only, you can create a configuration section of the form:

```
[url "<actual-url-base>"]
    pushInsteadOf = <other-url-base>
```

For example, with this:

```
[url "ssh://example.org/"]
```

```
pushInsteadOf = git://example.org/
```

a URL like "git://example.org/path/to/repo.git" will be rewritten to "ssh://example.org/path/to/repo.git" for pushes, but pulls will still use the original URL.

EXAMPLES

- Clone from upstream:

```
$ git clone git://git.kernel.org/pub/scm/.../linux.git my-linux
$ cd my-linux
$ make
```

- Make a local clone that borrows from the current directory, without checking things out:

```
$ git clone -l -s -n . ../copy
$ cd ../copy
$ git show-branch
```

- Clone from upstream while borrowing from an existing local directory:

```
$ git clone --reference /git/linux.git \
        git://git.kernel.org/pub/scm/.../linux.git \
        my-linux
$ cd my-linux
```

- Create a bare repository to publish your changes to the public:

```
$ git clone --bare -l /home/proj/.git /pub/scm/proj.git
```

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

init.templateDir

Specify the directory from which templates will be copied. (See the "TEMPLATE DIRECTORY" section of Section G.3.71, “git-init(1)”.)

init.defaultBranch

Allows overriding the default branch name e.g. when initializing a new repository.

clone.defaultRemoteName

The name of the remote to create when cloning a repository. Defaults to *origin*. It can be overridden by passing the *--origin* command-line option.

clone.rejectShallow

Reject cloning a repository if it is a shallow one; this can be overridden by passing the *--reject-shallow* option on the command line.

clone.filterSubmodules

If a partial clone filter is provided (see *--filter* in Section G.3.120, “git-rev-list(1)”) and *--recurse-submodules* is used, also apply the filter to submodules.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.25. git-column(1)

2024-07-02

Revision History

2024-07-02

NAME

git-column - Display data in columns

SYNOPSIS

git column [--command=<name>] [--[raw-]mode=<mode>] [--width=<width>] [--indent=<string>] [--nl=<string>] [--padding=<n>]

DESCRIPTION

This command formats the lines of its standard input into a table with multiple columns. Each input line occupies one cell of the table. It is used internally by other git commands to format output into columns.

OPTIONS

--command=<name>

Look up layout mode using configuration variable column.<name> and column.ui.

--mode=<mode>

Specify layout mode. See configuration variable column.ui for option syntax in Section G.3.29, “git-config(1)”.

--raw-mode=<n>

Same as --mode but take mode encoded as a number. This is mainly used by other commands that have already parsed layout mode.

--width=<width>

Specify the terminal width. By default *git column* will detect the terminal width, or fall back to 80 if it is unable to do so.

--indent=<string>

String to be printed at the beginning of each line.

--nl=<string>

String to be printed at the end of each line, including newline character.

--padding=<N>

The number of spaces between columns. One space by default.

EXAMPLES

Format data by columns:

```
$ seq 1 24 | git column --mode=column --padding=5
1       4       7       10      13      16      19      22
2       5       8       11      14      17      20      23
3       6       9       12      15      18      21      24
```

Format data by rows:

```
$ seq 1 21 | git column --mode=row --padding=5
1       2       3       4       5       6       7
8       9       10      11      12      13      14
15      16      17      18      19      20      21
```

List some tags in a table with unequal column widths:

```
$ git tag --list 'v2.4.*' --column=row,dense
v2.4.0  v2.4.0-rc0  v2.4.0-rc1  v2.4.0-rc2  v2.4.0-rc3
v2.4.1  v2.4.10     v2.4.11     v2.4.12     v2.4.2
v2.4.3  v2.4.4      v2.4.5      v2.4.6      v2.4.7
v2.4.8  v2.4.9
```

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

column.ui

Specify whether supported commands should output in columns. This variable consists of a list of tokens separated by spaces or commas:

These options control when the feature should be enabled (defaults to *never*):

always

always show in columns

never

never show in columns

auto

show in columns if the output is to the terminal

These options control layout (defaults to *column*). Setting any of these implies *always* if none of *always*, *never*, or *auto* are specified.

column

fill columns before rows

row

fill rows before columns

plain

show in one column

Finally, these options can be combined with a layout option (defaults to *nodense*):

dense

make unequal size columns to utilize more space

nodense

make equal size columns

column.branch

Specify whether to output branch listing in *git branch* in columns. See *column.ui* for details.

column.clean

Specify the layout when listing items in *git clean -i*, which always shows files and directories in columns. See *column.ui* for details.

column.status

Specify whether to output untracked files in *git status* in columns. See *column.ui* for details.

column.tag

Specify whether to output tag listings in *git tag* in columns. See *column.ui* for details.

GIT

Part of the Section G.3.1, "git(1)" suite

G.3.26. git-commit-graph(1)

2024-07-02
Revision History
2024-07-02

NAME

git-commit-graph - Write and verify Git commit-graph files

SYNOPSIS

git commit-graph verify [--object-dir <dir>] [--shallow] [--[no-]progress]
git commit-graph write [--object-dir <dir>] [--append]
[--split[=<strategy>]] [--reachable | --stdin-packs | --stdin-commits]
[--changed-paths] [--[no-]max-new-filters <n>] [--[no-]progress]
<split-options>

DESCRIPTION

Manage the serialized commit-graph file.

OPTIONS

--object-dir

Use given directory for the location of packfiles and commit-graph file. This parameter exists to specify the location of an alternate that only has the objects directory, not a full *.git* directory. The commit-graph file is expected to be in the *<dir>/info* directory and the packfiles are expected to be in *<dir>/pack*. If the directory could not be made into an absolute path, or does not match any known object directory, *git commit-graph ...* will exit with non-zero status.

--[no-]progress

Turn progress on/off explicitly. If neither is specified, progress is shown if standard error is connected to a terminal.

COMMANDS

write

Write a commit-graph file based on the commits found in packfiles. If the config option *core.commitGraph* is disabled, then this command will output a warning, then return success without writing a commit-graph file.

With the *--stdin-packs* option, generate the new commit graph by walking objects only in the specified pack-indexes. (Cannot be combined with *--stdin-commits* or *--reachable*.)

With the *--stdin-commits* option, generate the new commit graph by walking commits starting at the commits specified in stdin as a list of OIDs in hex, one OID per line. OIDs that resolve to non-commits (either directly, or by peeling tags) are silently ignored. OIDs that are malformed, or do not exist generate an error. (Cannot be combined with *--stdin-packs* or *--reachable*.)

With the *--reachable* option, generate the new commit graph by walking commits starting at all refs. (Cannot be combined with *--stdin-commits* or *--stdin-packs*.)

With the *--append* option, include all commits that are present in the existing commit-graph file.

With the *--changed-paths* option, compute and write information about the paths changed between a commit and its first parent. This operation can take a while on large repositories. It provides significant performance gains for getting history of a directory or a file with *git log -- <path>*. If this option is given, future commit-graph writes will automatically assume that this option was intended. Use *--no-changed-paths* to stop storing this data.

With the *--max-new-filters=<n>* option, generate at most *n* new Bloom filters (if *--changed-paths* is specified). If *n* is *-1*, no limit is enforced. Only commits present in the new layer count against this limit. To retroactively compute Bloom filters over earlier layers, it is advised to use *--split=replace*. Overrides the *commitGraph.maxNewFilters* configuration.

With the *--split[=<strategy>]* option, write the commit-graph as a chain of multiple commit-graph files stored in *<dir>/info/commit-graphs*. Commit-graph layers are merged based on the strategy and other splitting options. The new commits not already in the commit-graph are added in a new "tip" file. This file is merged with the existing file if the following merge conditions are met:

- If *--split=no-merge* is specified, a merge is never performed, and the remaining options are ignored. *--split=replace* overwrites the existing chain with a new one. A bare *--split* defers to the remaining options. (Note that merging a chain of commit graphs replaces the existing chain with a length-1 chain where the first and only incremental holds the entire graph).
- If *--size-multiple=<X>* is not specified, let *X* equal 2. If the new tip file would have *N* commits and the previous tip has *M* commits and *X* times *N* is greater than *M*, instead merge the two files into a single file.
- If *--max-commits=<M>* is specified with *M* a positive integer, and the new tip file would have more than *M* commits, then instead merge the new tip with the previous tip.

  Finally, if *--expire-time=<datetime>* is not specified, let *datetime* be the current time. After writing the split commit-graph, delete all unused commit-graph whose modified times are older than *datetime*.

verify

Read the commit-graph file and verify its contents against the object database. Used to check for corrupted data.

With the *--shallow* option, only check the tip commit-graph file in a chain of split commit-graphs.

EXAMPLES

- Write a commit-graph file for the packed commits in your local *.git* directory.

```
$ git commit-graph write
```

- Write a commit-graph file, extending the current commit-graph file using commits in *<pack-index>*.

```
$ echo <pack-index> | git commit-graph write --stdin-packs
```

- Write a commit-graph file containing all reachable commits.

```
$ git show-ref -s | git commit-graph write --stdin-commits
```

- Write a commit-graph file containing all commits in the current commit-graph file along with those reachable from *HEAD*.

```
$ git rev-parse HEAD | git commit-graph write --stdin-commits --append
```

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

commitGraph.generationVersion

Specifies the type of generation number version to use when writing or reading the commit-graph file. If version 1 is specified, then the corrected commit dates will not be written or read. Defaults to 2.

commitGraph.maxNewFilters

Specifies the default value for the *--max-new-filters* option of *git commit-graph write* (c.f., Section G.3.26, “git-commit-graph(1)”).

commitGraph.readChangedPaths

If true, then git will use the changed-path Bloom filters in the commit-graph file (if it exists, and they are present). Defaults to true. See Section G.3.26, “git-commit-graph(1)” for more information.

FILE FORMAT

see Section G.5.3, “gitformat-commit-graph(5)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.27. git-commit-tree(1)

2024-07-02

Revision History

2024-07-02

NAME

git-commit-tree - Create a new commit object

SYNOPSIS

git commit-tree <tree> [(-p <parent>)…]
git commit-tree [(-p <parent>)…] [-S[<keyid>]] [(-m <message>)…]
[(-F <file>)…] <tree>

DESCRIPTION

This is usually not what an end user wants to run directly. See Section G.3.28, “git-commit(1)” instead.

Creates a new commit object based on the provided tree object and emits the new commit object id on stdout. The log message is read from the standard input, unless *-m* or *-F* options are given.

The *-m* and *-F* options can be given any number of times, in any order. The commit log message will be composed in the order in which the options are given.

A commit object may have any number of parents. With exactly one parent, it is an ordinary commit. Having more than one parent makes the commit a merge between several lines of history. Initial (root) commits have no parents.

While a tree represents a particular directory state of a working directory, a commit represents that state in "time", and explains how to get there.

Normally a commit would identify a new "HEAD" state, and while Git doesn't care where you save the note about that state, in practice we tend to just write the result to the file that is pointed at by *.git/HEAD*, so that we can always see what the last committed state was.

OPTIONS

<tree>

An existing tree object.

-p <parent>

Each *-p* indicates the id of a parent commit object.

-m <message>

A paragraph in the commit log message. This can be given more than once and each <message> becomes its own paragraph.

-F <file>

Read the commit log message from the given file. Use - to read from the standard input. This can be given more than once and the content of each file becomes its own paragraph.

-S[<keyid>] , --gpg-sign[=<keyid>] , --no-gpg-sign

GPG-sign commits. The *keyid* argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space. *--no-gpg-sign* is useful to countermand a *--gpg-sign* option given earlier on the command line.

Commit Information

A commit encapsulates:

- all parent object ids
- author name, email and date
- committer name and email and the commit time.

A commit comment is read from stdin. If a changelog entry is not provided via "<" redirection, *git commit-tree* will just wait for one to be entered and terminated with ^D.

DATE FORMATS

The *GIT_AUTHOR_DATE* and *GIT_COMMITTER_DATE* environment variables support the following date formats:

Git internal format

It is *<unix-timestamp> <time-zone-offset>*, where *<unix-timestamp>* is the number of seconds since the UNIX epoch. *<time-zone-offset>* is a positive or negative offset from UTC. For example CET (which is 1 hour ahead of UTC) is *+0100*.

RFC 2822

The standard date format as described by RFC 2822, for example *Thu, 07 Apr 2005 22:13:13 +0200*.

ISO 8601

Time and date specified by the ISO 8601 standard, for example *2005-04-07T22:13:13*. The parser accepts a space instead of the *T* character as well. Fractional parts of a second will be ignored, for example *2005-04-07T22:13:13.019* will be treated as *2005-04-07T22:13:13*.

> **Note**
>
> In addition, the date part is accepted in the following formats: *YYYY.MM.DD*, *MM/DD/YYYY* and *DD.MM.YYYY*.

Discussion

Git is to some extent character encoding agnostic.

- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- Path names are encoded in UTF-8 normalization form C. This applies to tree objects, the index file, ref names, as well as path names in command line arguments, environment variables and config files (*.git/config* (see Section G.3.29, "git-config(1)"), Section G.4.5, "gitignore(5)", Section G.4.2, "gitattributes(5)" and Section G.4.10, "gitmodules(5)").

  Note that Git at the core level treats path names simply as sequences of non-NUL bytes, there are no path name encoding conversions (except on Mac and Windows). Therefore, using non-ASCII path names will mostly work even on platforms and file systems that use legacy extended ASCII encodings. However, repositories created on such systems will not work properly on UTF-8-based systems (e.g. Linux, Mac, Windows) and vice versa. Additionally, many Git-based tools simply assume path names to be UTF-8 and will fail to display other encodings correctly.
- Commit log messages are typically encoded in UTF-8, but other extended ASCII encodings are also supported. This includes ISO-8859-x, CP125x and many others, but *not* UTF-16/32, EBCDIC and CJK multi-byte encodings (GBK, Shift-JIS, Big5, EUC-x, CP9xx etc.).

Although we encourage that the commit log messages are encoded in UTF-8, both the core and Git Porcelain are designed not to force UTF-8 on projects. If all participants of a particular project find it more convenient to use legacy encodings, Git does not forbid it. However, there are a few things to keep in mind.

1. *git commit* and *git commit-tree* issue a warning if the commit log message given to it does not look like a valid UTF-8 string, unless you explicitly say your project uses a legacy encoding. The way to say this is to have *i18n.commitEncoding* in *.git/config* file, like this:

   ```
   [i18n]
           commitEncoding = ISO-8859-1
   ```

   Commit objects created with the above setting record the value of *i18n.commitEncoding* in their *encoding* header. This is to help other people who look at them later. Lack of this header implies that the commit log message is encoded in UTF-8.

2. *git log*, *git show*, *git blame* and friends look at the *encoding* header of a commit object, and try to re-code the log message into UTF-8 unless otherwise specified. You can specify the desired output encoding with *i18n.logOutputEncoding* in *.git/config* file, like this:

   ```
   [i18n]
           logOutputEncoding = ISO-8859-1
   ```

   If you do not have this configuration variable, the value of *i18n.commitEncoding* is used instead.

Note that we deliberately chose not to re-code the commit log message when a commit is made to force UTF-8 at the commit object level, because re-coding to UTF-8 is not necessarily a reversible operation.

FILES

/etc/mailname

SEE ALSO

Section G.3.160, “git-write-tree(1)” Section G.3.28, “git-commit(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.28. git-commit(1)

2024-07-02
Revision History
2024-07-02

NAME

git-commit - Record changes to the repository

SYNOPSIS

git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
[--dry-run] [(-c | -C | --squash) <commit> | --fixup [(amend|reword):]<commit>)]
[-F <file> | -m <msg>] [--reset-author] [--allow-empty]
[--allow-empty-message] [--no-verify] [-e] [--author=<author>]
[--date=<date>] [--cleanup=<mode>] [--[no-]status]
[-i | -o] [--pathspec-from-file=<file> [--pathspec-file-nul]]
[(--trailer <token>[(=|:)<value>])…] [-S[<keyid>]]
[--] [<pathspec>…]

DESCRIPTION

Create a new commit containing the current contents of the index and the given log message describing the changes. The new commit is a direct child of HEAD, usually the tip of the current branch, and the branch is updated to point to it (unless no branch is associated with the working tree, in which case HEAD is "detached" as described in Section G.3.19, “git-checkout(1)”).

The content to be committed can be specified in several ways:

1. by using Section G.3.2, “git-add(1)” to incrementally "add" changes to the index before using the *commit* command (Note: even modified files must be "added");

2. by using Section G.3.123, “git-rm(1)” to remove files from the working tree and the index, again before using the *commit* command;

3. by listing files as arguments to the *commit* command (without --interactive or --patch switch), in which case the commit will ignore changes staged in the index, and instead record the current content of the listed files (which must already be known to Git);

4. by using the -a switch with the *commit* command to automatically "add" changes from all known files (i.e. all files that are already listed in the index) and to automatically "rm" files in the index that have been removed from the working tree, and then perform the actual commit;

5. by using the --interactive or --patch switches with the *commit* command to decide one by one which files or hunks should be part of the commit in addition to contents in the index, before finalizing the operation. See the Interactive Mode section of Section G.3.2, “git-add(1)” to learn how to operate these modes.

The *--dry-run* option can be used to obtain a summary of what is included by any of the above for the next commit by giving the same set of parameters (options and paths).

If you make a commit and then find a mistake immediately after that, you can recover from it with *git reset*.

OPTIONS

-a , --all

Tell the command to automatically stage files that have been modified and deleted, but new files you have not told Git about are not affected.

-p , --patch

Use the interactive patch selection interface to choose which changes to commit. See Section G.3.2, “git-add(1)” for details.

-C <commit> , --reuse-message=<commit>

Take an existing commit object, and reuse the log message and the authorship information (including the timestamp) when creating the commit.

-c <commit> , --reedit-message=<commit>

Like *-C*, but with *-c* the editor is invoked, so that the user can further edit the commit message.

--fixup=[(amend|reword):]<commit>

Create a new commit which "fixes up" *<commit>* when applied with *git rebase --autosquash*. Plain *--fixup=<commit>* creates a "fixup!" commit which changes the content of *<commit>* but leaves its log message untouched. *--fixup=amend:<commit>* is similar but creates an "amend!" commit which also replaces the log message of *<commit>* with the log message of the "amend!" commit. *--fixup=reword:<commit>* creates an "amend!" commit which replaces the log message of *<commit>* with its own log message but makes no changes to the content of *<commit>*.

The commit created by plain *--fixup=<commit>* has a subject composed of "fixup!" followed by the subject line from <commit>, and is recognized specially by *git rebase --autosquash*. The *-m* option may be used to supplement the log message of the created commit, but the additional commentary will be thrown away once the "fixup!" commit is squashed into *<commit>* by *git rebase --autosquash*.

The commit created by *--fixup=amend:<commit>* is similar but its subject is instead prefixed with "amend!". The log message of <commit> is copied into the log message of the "amend!" commit and opened in an editor so it can be refined. When *git rebase --autosquash* squashes the "amend!" commit into *<commit>*, the log message of *<commit>* is replaced by the refined log message from the "amend!" commit. It is an error for the "amend!" commit's log message to be empty unless *--allow-empty-message* is specified.

--fixup=reword:<commit> is shorthand for *--fixup=amend:<commit> --only*. It creates an "amend!" commit with only a log message (ignoring any changes staged in the index). When squashed by *git rebase --autosquash*, it replaces the log message of *<commit>* without making any other changes.

Neither "fixup!" nor "amend!" commits change authorship of *<commit>* when applied by *git rebase --autosquash*. See Section G.3.107, “git-rebase(1)” for details.

--squash=<commit>

Construct a commit message for use with *rebase --autosquash*. The commit message subject line is taken from the specified commit with a prefix of "squash! ". Can be used with additional commit message options (*-m*/*-c*/*-C*/*-F*). See Section G.3.107, “git-rebase(1)” for details.

--reset-author

When used with -C/-c/--amend options, or when committing after a conflicting cherry-pick, declare that the authorship of the resulting commit now belongs to the committer. This also renews the author timestamp.

--short

When doing a dry-run, give the output in the short-format. See Section G.3.138, “git-status(1)” for details. Implies *--dry-run*.

--branch

Show the branch and tracking info even in short-format.

--porcelain

When doing a dry-run, give the output in a porcelain-ready format. See Section G.3.138, “git-status(1)” for details. Implies *--dry-run*.

--long

When doing a dry-run, give the output in the long-format. Implies *--dry-run*.

-z , --null

When showing *short* or *porcelain* status output, print the filename verbatim and terminate the entries with NUL, instead of LF. If no format is given, implies the *--porcelain* output format. Without the *-z* option, filenames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

-F <file> , --file=<file>

Take the commit message from the given file. Use - to read the message from the standard input.

--author=<author>

Override the commit author. Specify an explicit author using the standard *A U Thor <author@example.com>* format. Otherwise <author> is assumed to be a pattern and is used to search for an existing commit by that author (i.e. rev-list --all -i --author=<author>); the commit author is then copied from the first such commit found.

--date=<date>

Override the author date used in the commit.

-m <msg> , --message=<msg>

Use the given <msg> as the commit message. If multiple *-m* options are given, their values are concatenated as separate paragraphs.

The *-m* option is mutually exclusive with *-c*, *-C*, and *-F*.

-t <file> , --template=<file>

When editing the commit message, start the editor with the contents in the given file. The *commit.template* configuration variable is often used to give this option implicitly to the command. This mechanism can be used by projects that want to guide participants with some hints on what to write in the message in what order. If the user exits the editor without editing the message, the commit is aborted. This has no effect when a message is given by other means, e.g. with the *-m* or *-F* options.

-s , --signoff , --no-signoff

Add a *Signed-off-by* trailer by the committer at the end of the commit log message. The meaning of a signoff depends on the project to which you're committing. For example, it may certify that the committer has the rights to submit the work under the project's license or agrees to some contributor representation, such as a Developer Certificate of Origin. (See https://developercertificate.org for the one used by the Linux kernel and Git projects.) Consult the documentation or leadership of the project to which you're contributing to understand how the signoffs are used in that project.

The --no-signoff option can be used to countermand an earlier --signoff option on the command line.

--trailer <token>[(=|:)<value>]

Specify a (<token>, <value>) pair that should be applied as a trailer. (e.g. *git commit --trailer "Signed-off-by:C O Mitter \ <committer@example.com>" --trailer "Helped-by:C O Mitter \ <committer@example.com>"* will add the "Signed-off-by" trailer and the "Helped-by" trailer to the commit message.) The *trailer.** configuration variables (Section G.3.73, “git-interpret-trailers(1)”) can be used to define if a duplicated trailer is omitted, where in the run of trailers each trailer would appear, and other details.

-n , --[no-]verify

By default, the pre-commit and commit-msg hooks are run. When any of *--no-verify* or *-n* is given, these are bypassed. See also Section G.4.7, “githooks(5)”.

--allow-empty

Usually recording a commit that has the exact same tree as its sole parent commit is a mistake, and the command prevents you from making such a commit. This option bypasses the safety, and is primarily for use by foreign SCM interface scripts.

--allow-empty-message

Like --allow-empty this command is primarily for use by foreign SCM interface scripts. It allows you to create a commit with an empty commit message without using plumbing commands like Section G.3.27, “git-commit-tree(1)”.

--cleanup=<mode>

This option determines how the supplied commit message should be cleaned up before committing. The *<mode>* can be *strip*, *whitespace*, *verbatim*, *scissors* or *default*.

strip

Strip leading and trailing empty lines, trailing whitespace, commentary and collapse consecutive empty lines.

whitespace

Same as *strip* except #commentary is not removed.

verbatim

Do not change the message at all.

scissors

Same as *whitespace* except that everything from (and including) the line found below is truncated, if the message is to be edited. "#" can be customized with core.commentChar.

```
# ------------------------ >8 ------------------------
```

default

Same as *strip* if the message is to be edited. Otherwise *whitespace*.

The default can be changed by the *commit.cleanup* configuration variable (see Section G.3.29, “git-config(1)”).

-e , --edit

The message taken from file with *-F*, command line with *-m*, and from commit object with *-C* are usually used as the commit log message unmodified. This option lets you further edit the message taken from these sources.

--no-edit

Use the selected commit message without launching an editor. For example, *git commit --amend --no-edit* amends a commit without changing its commit message.

--amend

Replace the tip of the current branch by creating a new commit. The recorded tree is prepared as usual (including the effect of the *-i* and *-o* options and explicit pathspec), and the message from the original commit is used as the starting point, instead of an empty message, when no other message is specified from the command line via options such as *-m*, *-F*, *-c*, etc. The new commit has the same parents and author as the current one (the *--reset-author* option can countermand this).

It is a rough equivalent for:

```
$ git reset --soft HEAD^
$ ... do something else to come up with the right tree ...
$ git commit -c ORIG_HEAD
```

but can be used to amend a merge commit.

You should understand the implications of rewriting history if you amend a commit that has already been published. (See the "RECOVERING FROM UPSTREAM REBASE" section in Section G.3.107, “git-rebase(1)”.)

--no-post-rewrite

Bypass the post-rewrite hook.

-i , --include

Before making a commit out of staged contents so far, stage the contents of paths given on the command line as well. This is usually not what you want unless you are concluding a conflicted merge.

-o , --only

Make a commit by taking the updated working tree contents of the paths specified on the command line, disregarding any contents that have been staged for other paths. This is the default mode of operation of *git commit* if any paths are given on the command line, in which case this option can be omitted. If this option is specified together with *--amend*, then no paths need to be specified, which can be used to amend the last commit without committing changes that have already been staged. If used together with *--allow-empty* paths are also not required, and an empty commit will be created.

--pathspec-from-file=<file>

Pathspec is passed in *<file>* instead of commandline args. If *<file>* is exactly - then standard input is used. Pathspec elements are separated by LF or CR/LF. Pathspec elements can be quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). See also *--pathspec-file-nul* and global *--literal-pathspecs*.

--pathspec-file-nul

Only meaningful with *--pathspec-from-file*. Pathspec elements are separated with NUL character and all other characters are taken literally (including newlines and quotes).

-u[<mode>] , --untracked-files[=<mode>]

Show untracked files.

The mode parameter is optional (defaults to *all*), and is used to specify the handling of untracked files; when -u is not used, the default is *normal*, i.e. show untracked files and directories.

The possible options are:

- *no* - Show no untracked files
- *normal* - Shows untracked files and directories
- *all* - Also shows individual files in untracked directories.

All usual spellings for Boolean value *true* are taken as *normal* and *false* as *no*. The default can be changed using the status.showUntrackedFiles configuration variable documented in Section G.3.29, “git-config(1)”.

-v , --verbose

Show unified diff between the HEAD commit and what would be committed at the bottom of the commit message template to help the user describe the commit by reminding what changes the commit has. Note that this diff output doesn't have its lines prefixed with #. This diff will not be a part of the commit message. See the *commit.verbose* configuration variable in Section G.3.29, “git-config(1)”.

If specified twice, show in addition the unified diff between what would be committed and the worktree files, i.e. the unstaged changes to tracked files.

-q , --quiet

Suppress commit summary message.

--dry-run

Do not create a commit, but show a list of paths that are to be committed, paths with local changes that will be left uncommitted and paths that are untracked.

--status

Include the output of Section G.3.138, “git-status(1)” in the commit message template when using an editor to prepare the commit message. Defaults to on, but can be used to override configuration variable commit.status.

--no-status

Do not include the output of Section G.3.138, “git-status(1)” in the commit message template when using an editor to prepare the default commit message.

-S[<keyid>] , --gpg-sign[=<keyid>] , --no-gpg-sign

GPG-sign commits. The *keyid* argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space. *--no-gpg-sign* is useful to countermand both *commit.gpgSign* configuration variable, and earlier *--gpg-sign*.

--

Do not interpret any more arguments as options.

<pathspec>…

When pathspec is given on the command line, commit the contents of the files that match the pathspec without recording the changes already added to the index. The contents of these files are also staged for the next commit on top of what have been staged before.

For more details, see the *pathspec* entry in Section G.4.19, “gitglossary(7)”.

EXAMPLES

When recording your own work, the contents of modified files in your working tree are temporarily stored to a staging area called the "index" with *git add*. A file can be reverted back, only in the index but not in the working tree, to that of the last commit with *git restore --staged <file>*, which effectively reverts *git add* and prevents the changes to this file from participating in the next commit. After building the state to be committed incrementally

with these commands, *git commit* (without any pathname parameter) is used to record what has been staged so far. This is the most basic form of the command. An example:

```
$ edit hello.c
$ git rm goodbye.c
$ git add hello.c
$ git commit
```

Instead of staging files after each individual change, you can tell *git commit* to notice the changes to the files whose contents are tracked in your working tree and do corresponding *git add* and *git rm* for you. That is, this example does the same as the earlier example if there is no other change in your working tree:

```
$ edit hello.c
$ rm goodbye.c
$ git commit -a
```

The command *git commit -a* first looks at your working tree, notices that you have modified hello.c and removed goodbye.c, and performs necessary *git add* and *git rm* for you.

After staging changes to many files, you can alter the order the changes are recorded in, by giving pathnames to *git commit*. When pathnames are given, the command makes a commit that only records the changes made to the named paths:

```
$ edit hello.c hello.h
$ git add hello.c hello.h
$ edit Makefile
$ git commit Makefile
```

This makes a commit that records the modification to *Makefile*. The changes staged for *hello.c* and *hello.h* are not included in the resulting commit. However, their changes are not lost -- they are still staged and merely held back. After the above sequence, if you do:

```
$ git commit
```

this second commit would record the changes to *hello.c* and *hello.h* as expected.

After a merge (initiated by *git merge* or *git pull*) stops because of conflicts, cleanly merged paths are already staged to be committed for you, and paths that conflicted are left in unmerged state. You would have to first check which paths are conflicting with *git status* and after fixing them manually in your working tree, you would stage the result as usual with *git add*:

```
$ git status | grep unmerged
unmerged: hello.c
$ edit hello.c
$ git add hello.c
```

After resolving conflicts and staging the result, *git ls-files -u* would stop mentioning the conflicted path. When you are done, run *git commit* to finally record the merge:

```
$ git commit
```

As with the case to record your own changes, you can use *-a* option to save typing. One difference is that during a merge resolution, you cannot use *git commit* with pathnames to alter the order the changes are committed, because the merge should be recorded as a single commit. In fact, the command refuses to run when given pathnames (but see *-i* option).

COMMIT INFORMATION

Author and committer information is taken from the following environment variables, if set:

`GIT_AUTHOR_NAME`

```
GIT_AUTHOR_EMAIL
GIT_AUTHOR_DATE
GIT_COMMITTER_NAME
GIT_COMMITTER_EMAIL
GIT_COMMITTER_DATE
```

(nb "<", ">" and "\n"s are stripped)

The author and committer names are by convention some form of a personal name (that is, the name by which other humans refer to you), although Git does not enforce or require any particular form. Arbitrary Unicode may be used, subject to the constraints listed above. This name has no effect on authentication; for that, see the *credential.username* variable in Section G.3.29, “git-config(1)”.

In case (some of) these environment variables are not set, the information is taken from the configuration items *user.name* and *user.email*, or, if not present, the environment variable EMAIL, or, if that is not set, system user name and the hostname used for outgoing mail (taken from */etc/mailname* and falling back to the fully qualified hostname when that file does not exist).

The *author.name* and *committer.name* and their corresponding email options override *user.name* and *user.email* if set and are overridden themselves by the environment variables.

The typical usage is to set just the *user.name* and *user.email* variables; the other options are provided for more complex use cases.

DATE FORMATS

The *GIT_AUTHOR_DATE* and *GIT_COMMITTER_DATE* environment variables support the following date formats:

Git internal format

It is *<unix-timestamp> <time-zone-offset>*, where *<unix-timestamp>* is the number of seconds since the UNIX epoch. *<time-zone-offset>* is a positive or negative offset from UTC. For example CET (which is 1 hour ahead of UTC) is *+0100*.

RFC 2822

The standard date format as described by RFC 2822, for example *Thu, 07 Apr 2005 22:13:13 +0200*.

ISO 8601

Time and date specified by the ISO 8601 standard, for example *2005-04-07T22:13:13*. The parser accepts a space instead of the *T* character as well. Fractional parts of a second will be ignored, for example *2005-04-07T22:13:13.019* will be treated as *2005-04-07T22:13:13*.

> **Note**
>
> In addition, the date part is accepted in the following formats: *YYYY.MM.DD*, *MM/DD/YYYY* and *DD.MM.YYYY*.

In addition to recognizing all date formats above, the *--date* option will also try to make sense of other, more human-centric date formats, such as relative dates like "yesterday" or "last Friday at noon".

DISCUSSION

Though not required, it's a good idea to begin the commit message with a single short (no more than 50 characters) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout Git. For example, Section G.3.54, “git-format-patch(1)” turns a commit into email, and it uses the title on the Subject line and the rest of the commit in the body.

Git is to some extent character encoding agnostic.

- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- Path names are encoded in UTF-8 normalization form C. This applies to tree objects, the index file, ref names, as well as path names in command line arguments, environment variables and config files (*.git/config* (see Section G.3.29, "git-config(1)"), Section G.4.5, "gitignore(5)", Section G.4.2, "gitattributes(5)" and Section G.4.10, "gitmodules(5)").

  Note that Git at the core level treats path names simply as sequences of non-NUL bytes, there are no path name encoding conversions (except on Mac and Windows). Therefore, using non-ASCII path names will mostly work even on platforms and file systems that use legacy extended ASCII encodings. However, repositories created on such systems will not work properly on UTF-8-based systems (e.g. Linux, Mac, Windows) and vice versa. Additionally, many Git-based tools simply assume path names to be UTF-8 and will fail to display other encodings correctly.
- Commit log messages are typically encoded in UTF-8, but other extended ASCII encodings are also supported. This includes ISO-8859-x, CP125x and many others, but *not* UTF-16/32, EBCDIC and CJK multi-byte encodings (GBK, Shift-JIS, Big5, EUC-x, CP9xx etc.).

Although we encourage that the commit log messages are encoded in UTF-8, both the core and Git Porcelain are designed not to force UTF-8 on projects. If all participants of a particular project find it more convenient to use legacy encodings, Git does not forbid it. However, there are a few things to keep in mind.

1. *git commit* and *git commit-tree* issue a warning if the commit log message given to it does not look like a valid UTF-8 string, unless you explicitly say your project uses a legacy encoding. The way to say this is to have *i18n.commitEncoding* in *.git/config* file, like this:

   ```
   [i18n]
           commitEncoding = ISO-8859-1
   ```

   Commit objects created with the above setting record the value of *i18n.commitEncoding* in their *encoding* header. This is to help other people who look at them later. Lack of this header implies that the commit log message is encoded in UTF-8.

2. *git log*, *git show*, *git blame* and friends look at the *encoding* header of a commit object, and try to re-code the log message into UTF-8 unless otherwise specified. You can specify the desired output encoding with *i18n.logOutputEncoding* in *.git/config* file, like this:

   ```
   [i18n]
           logOutputEncoding = ISO-8859-1
   ```

   If you do not have this configuration variable, the value of *i18n.commitEncoding* is used instead.

Note that we deliberately chose not to re-code the commit log message when a commit is made to force UTF-8 at the commit object level, because re-coding to UTF-8 is not necessarily a reversible operation.

ENVIRONMENT AND CONFIGURATION VARIABLES

The editor used to edit the commit log message will be chosen from the *GIT_EDITOR* environment variable, the core.editor configuration variable, the *VISUAL* environment variable, or the *EDITOR* environment variable (in that order). See Section G.3.152, "git-var(1)" for details.

Everything above this line in this section isn't included from the Section G.3.29, "git-config(1)" documentation. The content that follows is the same as what's found there:

commit.cleanup

This setting overrides the default of the *--cleanup* option in *git commit*. See Section G.3.28, "git-commit(1)" for details. Changing the default can be useful when you always want to keep lines that begin with the comment

character # in your log message, in which case you would do *git config commit.cleanup whitespace* (note that you will have to remove the help lines that begin with # in the commit log template yourself, if you do this).

commit.gpgSign

A boolean to specify whether all commits should be GPG signed. Use of this option when doing operations such as rebase can result in a large number of commits being signed. It may be convenient to use an agent to avoid typing your GPG passphrase several times.

commit.status

A boolean to enable/disable inclusion of status information in the commit message template when using an editor to prepare the commit message. Defaults to true.

commit.template

Specify the pathname of a file to use as the template for new commit messages.

commit.verbose

A boolean or int to specify the level of verbosity with *git commit*. See Section G.3.28, “git-commit(1)”.

HOOKS

This command can run *commit-msg*, *prepare-commit-msg*, *pre-commit*, *post-commit* and *post-rewrite* hooks. See Section G.4.7, “githooks(5)” for more information.

FILES

$GIT_DIR/COMMIT_EDITMSG

This file contains the commit message of a commit in progress. If *git commit* exits due to an error before creating a commit, any commit message that has been provided by the user (e.g., in an editor session) will be available in this file, but will be overwritten by the next invocation of *git commit*.

SEE ALSO

Section G.3.2, “git-add(1)”, Section G.3.123, “git-rm(1)”, Section G.3.91, “git-mv(1)”, Section G.3.86, “git-merge(1)”, Section G.3.27, “git-commit-tree(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.29. git-config(1)

2024-07-02

Revision History

2024-07-02

NAME

git-config - Get and set repository or global options

SYNOPSIS

git config [<file-option>] [--type=<type>] [--comment=<message>] [--fixed-value] [--show-origin] [--show-scope] [-z|--n
git config [<file-option>] [--type=<type>] [--comment=<message>] --add <name> <value>
git config [<file-option>] [--type=<type>] [--comment=<message>] [--fixed-value] --replace-all <name> <value> [<value
git config [<file-option>] [--type=<type>] [--show-origin] [--show-scope] [-z|--null] [--fixed-value] --get <name> [<value
git config [<file-option>] [--type=<type>] [--show-origin] [--show-scope] [-z|--null] [--fixed-value] --get-all <name> [<va

git config [<file-option>] [--type=<type>] [--show-origin] [--show-scope] [-z|--null] [--fixed-value] [--name-only] --get-re
git config [<file-option>] [--type=<type>] [-z|--null] --get-urlmatch <name> <URL>
git config [<file-option>] [--fixed-value] --unset <name> [<value-pattern>]
git config [<file-option>] [--fixed-value] --unset-all <name> [<value-pattern>]
git config [<file-option>] --rename-section <old-name> <new-name>
git config [<file-option>] --remove-section <name>
git config [<file-option>] [--show-origin] [--show-scope] [-z|--null] [--name-only] -l | --list
git config [<file-option>] --get-color <name> [<default>]
git config [<file-option>] --get-colorbool <name> [<stdout-is-tty>]
git config [<file-option>] -e | --edit

DESCRIPTION

You can query/set/replace/unset options with this command. The name is actually the section and the key separated by a dot, and the value will be escaped.

Multiple lines can be added to an option by using the *--add* option. If you want to update or unset an option which can occur on multiple lines, a *value-pattern* (which is an extended regular expression, unless the *--fixed-value* option is given) needs to be given. Only the existing values that match the pattern are updated or unset. If you want to handle the lines that do **not** match the pattern, just prepend a single exclamation mark in front (see also the section called "EXAMPLES"), but note that this only works when the *--fixed-value* option is not in use.

The *--type=<type>* option instructs *git config* to ensure that incoming and outgoing values are canonicalize-able under the given <type>. If no *--type=<type>* is given, no canonicalization will be performed. Callers may unset an existing *--type* specifier with *--no-type*.

When reading, the values are read from the system, global and repository local configuration files by default, and options *--system*, *--global*, *--local*, *--worktree* and *--file <filename>* can be used to tell the command to read from only that location (see the section called "FILES").

When writing, the new value is written to the repository local configuration file by default, and options *--system*, *--global*, *--worktree*, *--file <filename>* can be used to tell the command to write to that location (you can say *--local* but that is the default).

This command will fail with non-zero status upon error. Some exit codes are:

- The section or key is invalid (ret=1),
- no section or name was provided (ret=2),
- the config file is invalid (ret=3),
- the config file cannot be written (ret=4),
- you try to unset an option which does not exist (ret=5),
- you try to unset/set an option for which multiple lines match (ret=5), or
- you try to use an invalid regexp (ret=6).

On success, the command returns the exit code 0.

A list of all available configuration variables can be obtained using the *git help --config* command.

OPTIONS

--replace-all

Default behavior is to replace at most one line. This replaces all lines matching the key (and optionally the *value-pattern*).

--add

Adds a new line to the option without altering any existing values. This is the same as providing ^$ as the *value-pattern* in *--replace-all*.

--comment <message>

Append a comment at the end of new or modified lines.

```
If _<message>_ begins with one or more whitespaces followed
by "#", it is used as-is.  If it begins with "#", a space is
prepended before it is used.  Otherwise, a string " # " (a
space followed by a hash followed by a space) is prepended
to it.  And the resulting string is placed immediately after
the value defined for the variable.  The _<message>_ must
not contain linefeed characters (no multi-line comments are
permitted).
```

--get

Get the value for a given key (optionally filtered by a regex matching the value). Returns error code 1 if the key was not found and the last value if multiple key values were found.

--get-all

Like get, but returns all values for a multi-valued key.

--get-regexp

Like --get-all, but interprets the name as a regular expression and writes out the key names. Regular expression matching is currently case-sensitive and done against a canonicalized version of the key in which section and variable names are lowercased, but subsection names are not.

--get-urlmatch <name> <URL>

When given a two-part <name> as <section>.<key>, the value for <section>.<URL>.<key> whose <URL> part matches the best to the given URL is returned (if no such key exists, the value for <section>.<key> is used as a fallback). When given just the <section> as name, do so for all the keys in the section and list them. Returns error code 1 if no value is found.

--global

For writing options: write to global *~/.gitconfig* file rather than the repository *.git/config*, write to *$XDG_CONFIG_HOME/git/config* file if this file exists and the *~/.gitconfig* file doesn't.

For reading options: read only from global *~/.gitconfig* and from *$XDG_CONFIG_HOME/git/config* rather than from all available files.

See also the section called “FILES”.

--system

For writing options: write to system-wide *$(prefix)/etc/gitconfig* rather than the repository *.git/config*.

For reading options: read only from system-wide *$(prefix)/etc/gitconfig* rather than from all available files.

See also the section called “FILES”.

--local

For writing options: write to the repository *.git/config* file. This is the default behavior.

For reading options: read only from the repository *.git/config* rather than from all available files.

See also the section called “FILES”.

--worktree

Similar to *--local* except that *$GIT_DIR/config.worktree* is read from or written to if *extensions.worktreeConfig* is enabled. If not it's the same as *--local*. Note that *$GIT_DIR* is equal to *$GIT_COMMON_DIR* for the main working tree, but is of the form *$GIT_DIR/worktrees/<id>/* for other working trees. See Section G.3.159, “git-worktree(1)” to learn how to enable *extensions.worktreeConfig*.

-f <config-file> , --file <config-file>

For writing options: write to the specified file rather than the repository *.git/config*.

For reading options: read only from the specified file rather than from all available files.

See also the section called “FILES”.

--blob <blob>

Similar to *--file* but use the given blob instead of a file. E.g. you can use *master:.gitmodules* to read values from the file *.gitmodules* in the master branch. See "SPECIFYING REVISIONS" section in Section G.4.14, “gitrevisions(7)” for a more complete list of ways to spell blob names.

--remove-section

Remove the given section from the configuration file.

--rename-section

Rename the given section to a new name.

--unset

Remove the line matching the key from config file.

--unset-all

Remove all lines matching the key from config file.

-l , --list

List all variables set in config file, along with their values.

--fixed-value

When used with the *value-pattern* argument, treat *value-pattern* as an exact string instead of a regular expression. This will restrict the name/value pairs that are matched to only those where the value is exactly equal to the *value-pattern*.

--type <type>

git config will ensure that any input or output is valid under the given type constraint(s), and will canonicalize outgoing values in *<type>*'s canonical form.

Valid *<type>*'s include:

- *bool*: canonicalize values as either "true" or "false".
- *int*: canonicalize values as simple decimal numbers. An optional suffix of *k*, *m*, or *g* will cause the value to be multiplied by 1024, 1048576, or 1073741824 upon input.

- *bool-or-int*: canonicalize according to either *bool* or *int*, as described above.
- *path*: canonicalize by expanding a leading ~ to the value of *$HOME* and *~user* to the home directory for the specified user. This specifier has no effect when setting the value (but you can use *git config section.variable ~/* from the command line to let your shell do the expansion.)
- *expiry-date*: canonicalize by converting from a fixed or relative date-string to a timestamp. This specifier has no effect when setting the value.
- *color*: When getting a value, canonicalize by converting to an ANSI color escape sequence. When setting a value, a sanity-check is performed to ensure that the given value is canonicalize-able as an ANSI color, but it is written as-is.

--bool , --int , --bool-or-int , --path , --expiry-date

Historical options for selecting a type specifier. Prefer instead *--type* (see above).

--no-type

Un-sets the previously set type specifier (if one was previously set). This option requests that *git config* not canonicalize the retrieved variable. *--no-type* has no effect without *--type=<type>* or *--<type>*.

-z , --null

For all options that output values and/or keys, always end values with the null character (instead of a newline). Use newline instead as a delimiter between key and value. This allows for secure parsing of the output without getting confused e.g. by values that contain line breaks.

--name-only

Output only the names of config variables for *--list* or *--get-regexp*.

--show-origin

Augment the output of all queried config options with the origin type (file, standard input, blob, command line) and the actual origin (config file path, ref, or blob id if applicable).

--show-scope

Similar to *--show-origin* in that it augments the output of all queried config options with the scope of that value (worktree, local, global, system, command).

--get-colorbool <name> [<stdout-is-tty>]

Find the color setting for *<name>* (e.g. *color.diff*) and output "true" or "false". *<stdout-is-tty>* should be either "true" or "false", and is taken into account when configuration says "auto". If *<stdout-is-tty>* is missing, then checks the standard output of the command itself, and exits with status 0 if color is to be used, or exits with status 1 otherwise. When the color setting for *name* is undefined, the command uses *color.ui* as fallback.

--get-color <name> [<default>]

Find the color configured for *name* (e.g. *color.diff.new*) and output it as the ANSI color escape sequence to the standard output. The optional *default* parameter is used instead, if there is no color configured for *name*.

--type=color [--default=<default>] is preferred over *--get-color* (but note that *--get-color* will omit the trailing newline printed by *--type=color*).

-e , --edit

Opens an editor to modify the specified config file; either *--system*, *--global*, *--local* (default), *--worktree*, or *--file <config-file>*.

--[no-]includes

Respect *include.** directives in config files when looking up values. Defaults to *off* when a specific file is given (e.g., using *--file*, *--global*, etc) and *on* when searching all config files.

--default <value>

When using *--get*, and the requested variable is not found, behave as if <value> were the value assigned to that variable.

CONFIGURATION

pager.config is only respected when listing configuration, i.e., when using *--list* or any of the *--get-** which may return multiple results. The default is to use a pager.

FILES

By default, *git config* will read configuration options from multiple files:

$(prefix)/etc/gitconfig

System-wide configuration file.

$XDG_CONFIG_HOME/git/config , ~/.gitconfig

User-specific configuration files. When the XDG_CONFIG_HOME environment variable is not set or empty, $HOME/.config/ is used as $XDG_CONFIG_HOME.

These are also called "global" configuration files. If both files exist, both files are read in the order given above.

$GIT_DIR/config

Repository specific configuration file.

$GIT_DIR/config.worktree

This is optional and is only searched when *extensions.worktreeConfig* is present in $GIT_DIR/config.

You may also provide additional configuration parameters when running any git command by using the *-c* option. See Section G.3.1, “git(1)” for details.

Options will be read from all of these files that are available. If the global or the system-wide configuration files are missing or unreadable they will be ignored. If the repository configuration file is missing or unreadable, *git config* will exit with a non-zero error code. An error message is produced if the file is unreadable, but not if it is missing.

The files are read in the order given above, with last value found taking precedence over values read earlier. When multiple values are taken then all values of a key from all files will be used.

By default, options are only written to the repository specific configuration file. Note that this also affects options like *--replace-all* and *--unset*. ***git config* will only ever change one file at a time**.

You can limit which configuration sources are read from or written to by specifying the path of a file with the *--file* option, or by specifying a configuration scope with *--system*, *--global*, *--local*, or *--worktree*. For more, see the section called “OPTIONS” above.

SCOPES

Each configuration source falls within a configuration scope. The scopes are:

system

$(prefix)/etc/gitconfig

global

$XDG_CONFIG_HOME/git/config

~/.gitconfig

local

$GIT_DIR/config

worktree

$GIT_DIR/config.worktree

command

GIT_CONFIG_{COUNT,KEY,VALUE} environment variables (see the section called “ENVIRONMENT” below)

the *-c* option

With the exception of *command*, each scope corresponds to a command line option: *--system*, *--global*, *--local*, *--worktree*.

When reading options, specifying a scope will only read options from the files within that scope. When writing options, specifying a scope will write to the files within that scope (instead of the repository specific configuration file). See the section called “OPTIONS” above for a complete description.

Most configuration options are respected regardless of the scope it is defined in, but some options are only respected in certain scopes. See the respective option's documentation for the full details.

1. Protected configuration

Protected configuration refers to the *system*, *global*, and *command* scopes. For security reasons, certain options are only respected when they are specified in protected configuration, and ignored otherwise.

Git treats these scopes as if they are controlled by the user or a trusted administrator. This is because an attacker who controls these scopes can do substantial harm without using Git, so it is assumed that the user's environment protects these scopes against attackers.

ENVIRONMENT

GIT_CONFIG_GLOBAL , GIT_CONFIG_SYSTEM

Take the configuration from the given files instead from global or system-level configuration. See Section G.3.1, “git(1)” for details.

GIT_CONFIG_NOSYSTEM

Whether to skip reading settings from the system-wide $(prefix)/etc/gitconfig file. See Section G.3.1, “git(1)” for details.

See also the section called “FILES”.

GIT_CONFIG_COUNT , GIT_CONFIG_KEY_<n> , GIT_CONFIG_VALUE_<n>

If GIT_CONFIG_COUNT is set to a positive number, all environment pairs GIT_CONFIG_KEY_<n> and GIT_CONFIG_VALUE_<n> up to that number will be added to the process's runtime configuration. The config pairs are zero-indexed. Any missing key or value is treated as an error. An empty GIT_CONFIG_COUNT is treated the same as GIT_CONFIG_COUNT=0, namely no pairs are processed. These environment variables will override values in configuration files, but will be overridden by any explicit options passed via *git -c*.

This is useful for cases where you want to spawn multiple git commands with a common configuration but cannot depend on a configuration file, for example when writing scripts.

GIT_CONFIG

If no *--file* option is provided to *git config*, use the file given by *GIT_CONFIG* as if it were provided via *--file*. This variable has no effect on other Git commands, and is mostly for historical compatibility; there is generally no reason to use it instead of the *--file* option.

EXAMPLES

Given a .git/config like this:

```
#
# This is the config file, and
# a '#' or ';' character indicates
# a comment
#

; core variables
[core]
        ; Don't trust file modes
        filemode = false

; Our diff algorithm
[diff]
        external = /usr/local/bin/diff-wrapper
        renames = true

; Proxy settings
[core]
        gitproxy=proxy-command for kernel.org
        gitproxy=default-proxy ; for all the rest

; HTTP
[http]
        sslVerify
[http "https://weak.example.com"]
        sslVerify = false
        cookieFile = /tmp/cookie.txt
```

you can set the filemode to true with

```
% git config core.filemode true
```

The hypothetical proxy command entries actually have a postfix to discern what URL they apply to. Here is how to change the entry for kernel.org to "ssh".

```
% git config core.gitproxy '"ssh" for kernel.org' 'for kernel.org$'
```

This makes sure that only the key/value pair for kernel.org is replaced.

To delete the entry for renames, do

```
% git config --unset diff.renames
```

If you want to delete an entry for a multivar (like core.gitproxy above), you have to provide a regex matching the value of exactly one line.

To query the value for a given key, do

```
% git config --get core.filemode
```

or

```
% git config core.filemode
```

or, to query a multivar:

```
% git config --get core.gitproxy "for kernel.org$"
```

If you want to know all the values for a multivar, do:

```
% git config --get-all core.gitproxy
```

If you like to live dangerously, you can replace **all** core.gitproxy by a new one with

```
% git config --replace-all core.gitproxy ssh
```

However, if you really only want to replace the line for the default proxy, i.e. the one without a "for …" postfix, do something like this:

```
% git config core.gitproxy ssh '! for '
```

To actually match only values with an exclamation mark, you have to

```
% git config section.key value '[!]'
```

To add a new proxy, without altering any of the existing ones, use

```
% git config --add core.gitproxy '"proxy-command" for example.com'
```

An example to use customized color from the configuration in your script:

```
#!/bin/sh
WS=$(git config --get-color color.diff.whitespace "blue reverse")
RESET=$(git config --get-color "" "reset")
echo "${WS}your whitespace color or blue reverse${RESET}"
```

For URLs in *https://weak.example.com*, *http.sslVerify* is set to false, while it is set to *true* for all others:

```
% git config --type=bool --get-urlmatch http.sslverify https://
good.example.com
true
% git config --type=bool --get-urlmatch http.sslverify https://
weak.example.com
false
% git config --get-urlmatch http https://weak.example.com
http.cookieFile /tmp/cookie.txt
http.sslverify false
```

CONFIGURATION FILE

The Git configuration file contains a number of variables that affect the Git commands' behavior. The files *.git/config* and optionally *config.worktree* (see the "CONFIGURATION FILE" section of Section G.3.159, “git-worktree(1)”) in each repository are used to store the configuration for that repository, and *$HOME/.gitconfig* is used to store a per-user configuration as fallback values for the *.git/config* file. The file */etc/gitconfig* can be used to store a system-wide default configuration.

The configuration variables are used by both the Git plumbing and the porcelain commands. The variables are divided into sections, wherein the fully qualified variable name of the variable itself is the last dot-separated segment and the section name is everything before the last dot. The variable names are case-insensitive, allow only

alphanumeric characters and -, and must start with an alphabetic character. Some variables may appear multiple times; we say then that the variable is multivalued.

1. Syntax

The syntax is fairly flexible and permissive. Whitespace characters, which in this context are the space character (SP) and the horizontal tabulation (HT), are mostly ignored. The *#* and *;* characters begin comments to the end of line. Blank lines are ignored.

The file consists of sections and variables. A section begins with the name of the section in square brackets and continues until the next section begins. Section names are case-insensitive. Only alphanumeric characters, - and . are allowed in section names. Each variable must belong to some section, which means that there must be a section header before the first setting of a variable.

Sections can be further divided into subsections. To begin a subsection put its name in double quotes, separated by space from the section name, in the section header, like in the example below:

```
[section "subsection"]
```

Subsection names are case sensitive and can contain any characters except newline and the null byte. Doublequote *"* and backslash can be included by escaping them as *\"* and **, respectively. Backslashes preceding other characters are dropped when reading; for example, *\t* is read as *t* and *\0* is read as *0*. Section headers cannot span multiple lines. Variables may belong directly to a section or to a given subsection. You can have *[section]* if you have *[section "subsection"]*, but you don't need to.

There is also a deprecated *[section.subsection]* syntax. With this syntax, the subsection name is converted to lower-case and is also compared case sensitively. These subsection names follow the same restrictions as section names.

All the other lines (and the remainder of the line after the section header) are recognized as setting variables, in the form *name = value* (or just *name*, which is a short-hand to say that the variable is the boolean "true"). The variable names are case-insensitive, allow only alphanumeric characters and -, and must start with an alphabetic character.

Whitespace characters surrounding *name*, = and *value* are discarded. Internal whitespace characters within *value* are retained verbatim. Comments starting with either *#* or *;* and extending to the end of line are discarded. A line that defines a value can be continued to the next line by ending it with a backslash (\); the backslash and the end-of-line characters are discarded.

If *value* needs to contain leading or trailing whitespace characters, it must be enclosed in double quotation marks (*"*). Inside double quotation marks, double quote (*"*) and backslash (\) characters must be escaped: use *\"* for *"* and ** for **.

The following escape sequences (beside *\"* and **) are recognized: *\n* for newline character (NL), *\t* for horizontal tabulation (HT, TAB) and *\b* for backspace (BS). Other char escape sequences (including octal escape sequences) are invalid.

2. Includes

The *include* and *includeIf* sections allow you to include config directives from another source. These sections behave identically to each other with the exception that *includeIf* sections may be ignored if their condition does not evaluate to true; see "Conditional includes" below.

You can include a config file from another by setting the special *include.path* (or *includeIf.*.path*) variable to the name of the file to be included. The variable takes a pathname as its value, and is subject to tilde expansion. These variables can be given multiple times.

The contents of the included file are inserted immediately, as if they had been found at the location of the include directive. If the value of the variable is a relative path, the path is considered to be relative to the configuration file in which the include directive was found. See below for examples.

3. Conditional includes

You can conditionally include a config file from another by setting an *includeIf.<condition>.path* variable to the name of the file to be included.

The condition starts with a keyword followed by a colon and some data whose format and meaning depends on the keyword. Supported keywords are:

gitdir

The data that follows the keyword *gitdir:* is used as a glob pattern. If the location of the .git directory matches the pattern, the include condition is met.

The .git location may be auto-discovered, or come from *$GIT_DIR* environment variable. If the repository is auto-discovered via a .git file (e.g. from submodules, or a linked worktree), the .git location would be the final location where the .git directory is, not where the .git file is.

The pattern can contain standard globbing wildcards and two additional ones, **/ and /**, that can match multiple path components. Please refer to Section G.4.5, “gitignore(5)” for details. For convenience:

- If the pattern starts with ~/, ~ will be substituted with the content of the environment variable *HOME*.
- If the pattern starts with ./, it is replaced with the directory containing the current config file.
- If the pattern does not start with either ~/, ./ or /, **/ will be automatically prepended. For example, the pattern *foo/bar* becomes **/foo/bar and would match */any/path/to/foo/bar*.
- If the pattern ends with /, ** will be automatically added. For example, the pattern *foo/* becomes *foo/**. In other words, it matches "foo" and everything inside, recursively.

gitdir/i

This is the same as *gitdir* except that matching is done case-insensitively (e.g. on case-insensitive file systems)

onbranch

The data that follows the keyword *onbranch:* is taken to be a pattern with standard globbing wildcards and two additional ones, **/ and /**, that can match multiple path components. If we are in a worktree where the name of the branch that is currently checked out matches the pattern, the include condition is met.

If the pattern ends with /, ** will be automatically added. For example, the pattern *foo/* becomes *foo/**. In other words, it matches all branches that begin with *foo/*. This is useful if your branches are organized hierarchically and you would like to apply a configuration to all the branches in that hierarchy.

hasconfig:remote..url:*

The data that follows this keyword is taken to be a pattern with standard globbing wildcards and two additional ones, **/ and /**, that can match multiple components. The first time this keyword is seen, the rest of the config files will be scanned for remote URLs (without applying any values). If there exists at least one remote URL that matches this pattern, the include condition is met.

Files included by this option (directly or indirectly) are not allowed to contain remote URLs.

Note that unlike other includeIf conditions, resolving this condition relies on information that is not yet known at the point of reading the condition. A typical use case is this option being present as a system-level or global-level config, and the remote URL being in a local-level config; hence the need to scan ahead when resolving this condition. In order to avoid the chicken-and-egg problem in which potentially-included files can affect whether such files are potentially included, Git breaks the cycle by prohibiting these files from affecting the resolution of these conditions (thus, prohibiting them from declaring remote URLs).

As for the naming of this keyword, it is for forwards compatibility with a naming scheme that supports more variable-based include conditions, but currently Git only supports the exact keyword described above.

A few more notes on matching via *gitdir* and *gitdir/i*:

- Symlinks in *$GIT_DIR* are not resolved before matching.
- Both the symlink & realpath versions of paths will be matched outside of *$GIT_DIR*. E.g. if ~/git is a symlink to /mnt/storage/git, both *gitdir:~/git* and *gitdir:/mnt/storage/git* will match.

  This was not the case in the initial release of this feature in v2.13.0, which only matched the realpath version. Configuration that wants to be compatible with the initial release of this feature needs to either specify only the realpath version, or both versions.
- Note that "../" is not special and will match literally, which is unlikely what you want.

4. Example

```
# Core variables
[core]
        ; Don't trust file modes
        filemode = false

# Our diff algorithm
[diff]
        external = /usr/local/bin/diff-wrapper
        renames = true

[branch "devel"]
        remote = origin
        merge = refs/heads/devel

# Proxy settings
[core]
        gitProxy="ssh" for "kernel.org"
        gitProxy=default-proxy ; for the rest

[include]
        path = /path/to/foo.inc ; include by absolute path
        path = foo.inc ; find "foo.inc" relative to the current file
        path = ~/foo.inc ; find "foo.inc" in your `$HOME` directory

; include if $GIT_DIR is /path/to/foo/.git
[includeIf "gitdir:/path/to/foo/.git"]
        path = /path/to/foo.inc

; include for all repositories inside /path/to/group
[includeIf "gitdir:/path/to/group/"]
        path = /path/to/foo.inc

; include for all repositories inside $HOME/to/group
[includeIf "gitdir:~/to/group/"]
        path = /path/to/foo.inc

; relative paths are always relative to the including
; file (if the condition is true); their location is not
; affected by the condition
[includeIf "gitdir:/path/to/group/"]
        path = foo.inc

; include only if we are in a worktree where foo-branch is
; currently checked out
```

```
[includeIf "onbranch:foo-branch"]
        path = foo.inc

; include only if a remote with the given URL exists (note
; that such a URL may be provided later in a file or in a
; file read after this file is read, as seen in this example)
[includeIf "hasconfig:remote.*.url:https://example.com/**"]
        path = foo.inc
[remote "origin"]
        url = https://example.com/git
```

5. Values

Values of many variables are treated as a simple string, but there are variables that take values of specific types and there are rules as to how to spell them.

boolean

When a variable is said to take a boolean value, many synonyms are accepted for *true* and *false*; these are all case-insensitive.

true

Boolean true literals are *yes*, *on*, *true*, and *1*. Also, a variable defined without = <*value*> is taken as true.

false

Boolean false literals are *no*, *off*, *false*, *0* and the empty string.

When converting a value to its canonical form using the *--type=bool* type specifier, *git config* will ensure that the output is "true" or "false" (spelled in lowercase).

integer

The value for many variables that specify various sizes can be suffixed with *k*, *M*,… to mean "scale the number by 1024", "by 1024x1024", etc.

color

The value for a variable that takes a color is a list of colors (at most two, one for foreground and one for background) and attributes (as many as you want), separated by spaces.

The basic colors accepted are *normal*, *black*, *red*, *green*, *yellow*, *blue*, *magenta*, *cyan*, *white* and *default*. The first color given is the foreground; the second is the background. All the basic colors except *normal* and *default* have a bright variant that can be specified by prefixing the color with *bright*, like *brightred*.

The color *normal* makes no change to the color. It is the same as an empty string, but can be used as the foreground color when specifying a background color alone (for example, "normal red").

The color *default* explicitly resets the color to the terminal default, for example to specify a cleared background. Although it varies between terminals, this is usually not the same as setting to "white black".

Colors may also be given as numbers between 0 and 255; these use ANSI 256-color mode (but note that not all terminals may support this). If your terminal supports it, you may also specify 24-bit RGB values as hex, like *#ff0ab3*.

The accepted attributes are *bold*, *dim*, *ul*, *blink*, *reverse*, *italic*, and *strike* (for crossed-out or "strikethrough" letters). The position of any attributes with respect to the colors (before, after, or in between), doesn't matter. Specific attributes may be turned off by prefixing them with *no* or *no-* (e.g., *noreverse*, *no-ul*, etc).

The pseudo-attribute *reset* resets all colors and attributes before applying the specified coloring. For example, *reset green* will result in a green foreground and default background without any active attributes.

An empty color string produces no color effect at all. This can be used to avoid coloring specific elements without disabling color entirely.

For git's pre-defined color slots, the attributes are meant to be reset at the beginning of each item in the colored output. So setting *color.decorate.branch* to *black* will paint that branch name in a plain *black*, even if the previous thing on the same output line (e.g. opening parenthesis before the list of branch names in *log --decorate* output) is set to be painted with *bold* or some other attribute. However, custom log formats may do more complicated and layered coloring, and the negated forms may be useful there.

pathname

A variable that takes a pathname value can be given a string that begins with "~/" or "*~user/*", and the usual tilde expansion happens to such a string: *~/* is expanded to the value of *$HOME*, and *~user/* to the specified user's home directory.

If a path starts with *%(prefix)/*, the remainder is interpreted as a path relative to Git's "runtime prefix", i.e. relative to the location where Git itself was installed. For example, *%(prefix)/bin/* refers to the directory in which the Git executable itself lives. If Git was compiled without runtime prefix support, the compiled-in prefix will be substituted instead. In the unlikely event that a literal path needs to be specified that should *not* be expanded, it needs to be prefixed by *./*, like so: *./%(prefix)/bin*.

6. Variables

Note that this list is non-comprehensive and not necessarily complete. For command-specific variables, you will find a more detailed description in the appropriate manual page.

Other git-related tools may and do use their own variables. When inventing new variables for use in your own tool, make sure their names do not conflict with those that are used by Git itself and other popular tools, and describe them in your documentation.

add.ignoreErrors , add.ignore-errors (deprecated)

Tells *git add* to continue adding files when some files cannot be added due to indexing errors. Equivalent to the *--ignore-errors* option of Section G.3.2, “git-add(1)”. *add.ignore-errors* is deprecated, as it does not follow the usual naming convention for configuration variables.

add.interactive.useBuiltin

Unused configuration variable. Used in Git versions v2.25.0 to v2.36.0 to enable the built-in version of Section G.3.2, “git-add(1)”'s interactive mode, which then became the default in Git versions v2.37.0 to v2.39.0.

advice.*

These variables control various optional help messages designed to aid new users. When left unconfigured, Git will give the message alongside instructions on how to squelch it. You can tell Git that you do not need the help message by setting these to *false*:

addEmbeddedRepo

Shown when the user accidentally adds one git repo inside of another.

addEmptyPathspec

Shown when the user runs *git add* without providing the pathspec parameter.

addIgnoredFile

Shown when the user attempts to add an ignored file to the index.

amWorkDir

Shown when Section G.3.3, “git-am(1)” fails to apply a patch file, to tell the user the location of the file.

ambiguousFetchRefspec

Shown when a fetch refspec for multiple remotes maps to the same remote-tracking branch namespace and causes branch tracking set-up to fail.

checkoutAmbiguousRemoteBranchName

Shown when the argument to Section G.3.19, “git-checkout(1)” and Section G.3.140, “git-switch(1)” ambiguously resolves to a remote tracking branch on more than one remote in situations where an unambiguous argument would have otherwise caused a remote-tracking branch to be checked out. See the *checkout.defaultRemote* configuration variable for how to set a given remote to be used by default in some situations where this advice would be printed.

commitBeforeMerge

Shown when Section G.3.86, “git-merge(1)” refuses to merge to avoid overwriting local changes.

detachedHead

Shown when the user uses Section G.3.140, “git-switch(1)” or Section G.3.19, “git-checkout(1)” to move to the detached HEAD state, to tell the user how to create a local branch after the fact.

diverging

Shown when a fast-forward is not possible.

fetchShowForcedUpdates

Shown when Section G.3.49, “git-fetch(1)” takes a long time to calculate forced updates after ref updates, or to warn that the check is disabled.

forceDeleteBranch

Shown when the user tries to delete a not fully merged branch without the force option set.

ignoredHook

Shown when a hook is ignored because the hook is not set as executable.

implicitIdentity

Shown when the user's information is guessed from the system username and domain name, to tell the user how to set their identity configuration.

mergeConflict

Shown when various commands stop because of conflicts.

nestedTag

Shown when a user attempts to recursively tag a tag object.

pushAlreadyExists

Shown when Section G.3.103, “git-push(1)” rejects an update that does not qualify for fast-forwarding (e.g., a tag.)

pushFetchFirst

Shown when Section G.3.103, “git-push(1)” rejects an update that tries to overwrite a remote ref that points at an object we do not have.

pushNeedsForce

Shown when Section G.3.103, “git-push(1)” rejects an update that tries to overwrite a remote ref that points at an object that is not a commit-ish, or make the remote ref point at an object that is not a commit-ish.

pushNonFFCurrent

Shown when Section G.3.103, “git-push(1)” fails due to a non-fast-forward update to the current branch.

pushNonFFMatching

Shown when the user ran Section G.3.103, “git-push(1)” and pushed "matching refs" explicitly (i.e. used *:*, or specified a refspec that isn't the current branch) and it resulted in a non-fast-forward error.

pushRefNeedsUpdate

Shown when Section G.3.103, “git-push(1)” rejects a forced update of a branch when its remote-tracking ref has updates that we do not have locally.

pushUnqualifiedRefname

Shown when Section G.3.103, “git-push(1)” gives up trying to guess based on the source and destination refs what remote ref namespace the source belongs in, but where we can still suggest that the user push to either *refs/heads/** or *refs/tags/** based on the type of the source object.

pushUpdateRejected

Set this variable to *false* if you want to disable *pushNonFFCurrent*, *pushNonFFMatching*, *pushAlreadyExists*, *pushFetchFirst*, *pushNeedsForce*, and *pushRefNeedsUpdate* simultaneously.

rebaseTodoError

Shown when there is an error after editing the rebase todo list.

refSyntax

Shown when the user provides an illegal ref name, to tell the user about the ref syntax documentation.

resetNoRefresh

Shown when Section G.3.118, “git-reset(1)” takes more than 2 seconds to refresh the index after reset, to tell the user that they can use the *--no-refresh* option.

resolveConflict

Shown by various commands when conflicts prevent the operation from being performed.

rmHints

Shown on failure in the output of Section G.3.123, “git-rm(1)”, to give directions on how to proceed from the current state.

sequencerInUse

Shown when a sequencer command is already in progress.

skippedCherryPicks

Shown when Section G.3.107, “git-rebase(1)” skips a commit that has already been cherry-picked onto the upstream branch.

statusAheadBehind

Shown when Section G.3.138, “git-status(1)” computes the ahead/behind counts for a local ref compared to its remote tracking ref, and that calculation takes longer than expected. Will not appear if *status.aheadBehind* is false or the option *--no-ahead-behind* is given.

statusHints

Show directions on how to proceed from the current state in the output of Section G.3.138, “git-status(1)”, in the template shown when writing commit messages in Section G.3.28, “git-commit(1)”, and in the help message shown by Section G.3.140, “git-switch(1)” or Section G.3.19, “git-checkout(1)” when switching branches.

statusUoption

Shown when Section G.3.138, “git-status(1)” takes more than 2 seconds to enumerate untracked files, to tell the user that they can use the *-u* option.

submoduleAlternateErrorStrategyDie

Shown when a submodule.alternateErrorStrategy option configured to "die" causes a fatal error.

submoduleMergeConflict

Advice shown when a non-trivial submodule merge conflict is encountered.

submodulesNotUpdated

Shown when a user runs a submodule command that fails because *git submodule update --init* was not run.

suggestDetachingHead

Shown when Section G.3.140, “git-switch(1)” refuses to detach HEAD without the explicit *--detach* option.

updateSparsePath

Shown when either Section G.3.2, “git-add(1)” or Section G.3.123, “git-rm(1)” is asked to update index entries outside the current sparse checkout.

waitingForEditor

Shown when Git is waiting for editor input. Relevant when e.g. the editor is not launched inside the terminal.

worktreeAddOrphan

Shown when the user tries to create a worktree from an invalid reference, to tell the user how to create a new unborn branch instead.

alias.*

Command aliases for the Section G.3.1, “git(1)” command wrapper - e.g. after defining *alias.last = cat-file commit HEAD*, the invocation *git last* is equivalent to *git cat-file commit HEAD*. To avoid confusion and troubles with script usage, aliases that hide existing Git commands are ignored. Arguments are split by spaces, the usual shell quoting and escaping are supported. A quote pair or a backslash can be used to quote them.

Note that the first word of an alias does not necessarily have to be a command. It can be a command-line option that will be passed into the invocation of *git*. In particular, this is useful when used with *-c* to pass in one-time configurations or *-p* to force pagination. For example, *loud-rebase = -c commit.verbose=true rebase*

can be defined such that running *git loud-rebase* would be equivalent to *git -c commit.verbose=true rebase*. Also, *ps = -p status* would be a helpful alias since *git ps* would paginate the output of *git status* where the original command does not.

If the alias expansion is prefixed with an exclamation point, it will be treated as a shell command. For example, defining *alias.new = !gitk --all --not ORIG_HEAD*, the invocation *git new* is equivalent to running the shell command *gitk --all --not ORIG_HEAD*. Note that shell commands will be executed from the top-level directory of a repository, which may not necessarily be the current directory. *GIT_PREFIX* is set as returned by running *git rev-parse --show-prefix* from the original current directory. See Section G.3.121, "git-rev-parse(1)".

am.keepcr

If true, git-am will call git-mailsplit for patches in mbox format with parameter *--keep-cr*. In this case git-mailsplit will not remove *\r* from lines ending with *\r\n*. Can be overridden by giving *--no-keep-cr* from the command line. See Section G.3.3, "git-am(1)", Section G.3.79, "git-mailsplit(1)".

am.threeWay

By default, *git am* will fail if the patch does not apply cleanly. When set to true, this setting tells *git am* to fall back on 3-way merge if the patch records the identity of blobs it is supposed to apply to and we have those blobs available locally (equivalent to giving the *--3way* option from the command line). Defaults to *false*. See Section G.3.3, "git-am(1)".

apply.ignoreWhitespace

When set to *change*, tells *git apply* to ignore changes in whitespace, in the same way as the *--ignore-space-change* option. When set to one of: no, none, never, false, it tells *git apply* to respect all whitespace differences. See Section G.3.5, "git-apply(1)".

apply.whitespace

Tells *git apply* how to handle whitespace, in the same way as the *--whitespace* option. See Section G.3.5, "git-apply(1)".

attr.tree

A reference to a tree in the repository from which to read attributes, instead of the *.gitattributes* file in the working tree. If the value does not resolve to a valid tree object, an empty tree is used instead. When the *GIT_ATTR_SOURCE* environment variable or *--attr-source* command line option are used, this configuration variable has no effect.

blame.blankBoundary

Show blank commit object name for boundary commits in Section G.3.9, "git-blame(1)". This option defaults to false.

blame.coloring

This determines the coloring scheme to be applied to blame output. It can be *repeatedLines*, *highlightRecent*, or *none* which is the default.

blame.date

Specifies the format used to output dates in Section G.3.9, "git-blame(1)". If unset the iso format is used. For supported values, see the discussion of the *--date* option at Section G.3.74, "git-log(1)".

blame.showEmail

Show the author email instead of author name in Section G.3.9, "git-blame(1)". This option defaults to false.

blame.showRoot

Do not treat root commits as boundaries in Section G.3.9, “git-blame(1)”. This option defaults to false.

blame.ignoreRevsFile

Ignore revisions listed in the file, one unabbreviated object name per line, in Section G.3.9, “git-blame(1)”. Whitespace and comments beginning with # are ignored. This option may be repeated multiple times. Empty file names will reset the list of ignored revisions. This option will be handled before the command line option *--ignore-revs-file*.

blame.markUnblamableLines

Mark lines that were changed by an ignored revision that we could not attribute to another commit with a * in the output of Section G.3.9, “git-blame(1)”.

blame.markIgnoredLines

Mark lines that were changed by an ignored revision that we attributed to another commit with a *?* in the output of Section G.3.9, “git-blame(1)”.

branch.autoSetupMerge

Tells *git branch*, *git switch* and *git checkout* to set up new branches so that Section G.3.102, “git-pull(1)” will appropriately merge from the starting point branch. Note that even if this option is not set, this behavior can be chosen per-branch using the *--track* and *--no-track* options. The valid settings are: *false* -- no automatic setup is done; *true* -- automatic setup is done when the starting point is a remote-tracking branch; *always* -- automatic setup is done when the starting point is either a local branch or remote-tracking branch; *inherit* -- if the starting point has a tracking configuration, it is copied to the new branch; *simple* -- automatic setup is done only when the starting point is a remote-tracking branch and the new branch has the same name as the remote branch. This option defaults to true.

branch.autoSetupRebase

When a new branch is created with *git branch*, *git switch* or *git checkout* that tracks another branch, this variable tells Git to set up pull to rebase instead of merge (see "branch.<name>.rebase"). When *never*, rebase is never automatically set to true. When *local*, rebase is set to true for tracked branches of other local branches. When *remote*, rebase is set to true for tracked branches of remote-tracking branches. When *always*, rebase will be set to true for all tracking branches. See "branch.autoSetupMerge" for details on how to set up a branch to track another branch. This option defaults to never.

branch.sort

This variable controls the sort ordering of branches when displayed by Section G.3.10, “git-branch(1)”. Without the "--sort=<value>" option provided, the value of this variable will be used as the default. See Section G.3.52, “git-for-each-ref(1)” field names for valid values.

branch.<name>.remote

When on branch <name>, it tells *git fetch* and *git push* which remote to fetch from or push to. The remote to push to may be overridden with *remote.pushDefault* (for all branches). The remote to push to, for the current branch, may be further overridden by *branch.<name>.pushRemote*. If no remote is configured, or if you are not on any branch and there is more than one remote defined in the repository, it defaults to *origin* for fetching and *remote.pushDefault* for pushing. Additionally, . (a period) is the current local repository (a dot-repository), see *branch.<name>.merge*'s final note below.

branch.<name>.pushRemote

When on branch <name>, it overrides *branch.<name>.remote* for pushing. It also overrides *remote.pushDefault* for pushing from branch <name>. When you pull from one place (e.g. your upstream) and push to an-

other place (e.g. your own publishing repository), you would want to set *remote.pushDefault* to specify the remote to push to for all branches, and use this option to override it for a specific branch.

branch.<name>.merge

Defines, together with branch.<name>.remote, the upstream branch for the given branch. It tells *git fetch*/*git pull*/*git rebase* which branch to merge and can also affect *git push* (see push.default). When in branch <name>, it tells *git fetch* the default refspec to be marked for merging in FETCH_HEAD. The value is handled like the remote part of a refspec, and must match a ref which is fetched from the remote given by "branch.<name>.remote". The merge information is used by *git pull* (which first calls *git fetch*) to lookup the default branch for merging. Without this option, *git pull* defaults to merge the first refspec fetched. Specify multiple values to get an octopus merge. If you wish to setup *git pull* so that it merges into <name> from another branch in the local repository, you can point branch.<name>.merge to the desired branch, and use the relative path setting . (a period) for branch.<name>.remote.

branch.<name>.mergeOptions

Sets default options for merging into branch <name>. The syntax and supported options are the same as those of Section G.3.86, “git-merge(1)”, but option values containing whitespace characters are currently not supported.

branch.<name>.rebase

When true, rebase the branch <name> on top of the fetched branch, instead of merging the default branch from the default remote when "git pull" is run. See "pull.rebase" for doing this in a non branch-specific manner.

When *merges* (or just *m*), pass the *--rebase-merges* option to *git rebase* so that the local merge commits are included in the rebase (see Section G.3.107, “git-rebase(1)” for details).

When the value is *interactive* (or just *i*), the rebase is run in interactive mode.

NOTE: this is a possibly dangerous operation; do **not** use it unless you understand the implications (see Section G.3.107, “git-rebase(1)” for details).

branch.<name>.description

Branch description, can be edited with *git branch --edit-description*. Branch description is automatically added to the format-patch cover letter or request-pull summary.

browser.<tool>.cmd

Specify the command to invoke the specified browser. The specified command is evaluated in shell with the URLs passed as arguments. (See Section G.3.157, “git-web--browse(1)”.)

browser.<tool>.path

Override the path for the given tool that may be used to browse HTML help (see *-w* option in Section G.3.63, “git-help(1)”) or a working repository in gitweb (see Section G.3.72, “git-instaweb(1)”).

bundle.*

The *bundle.** keys may appear in a bundle list file found via the *git clone --bundle-uri* option. These keys currently have no effect if placed in a repository config file, though this will change in the future. See *the bundle URI design document* [https://www.kernel.org/pub/software/scm/git/docs/technical/bundle-uri.html] for more details.

bundle.version

This integer value advertises the version of the bundle list format used by the bundle list. Currently, the only accepted value is *1*.

bundle.mode

This string value should be either *all* or *any*. This value describes whether all of the advertised bundles are required to unbundle a complete understanding of the bundled information (*all*) or if any one of the listed bundle URIs is sufficient (*any*).

bundle.heuristic

If this string-valued key exists, then the bundle list is designed to work well with incremental *git fetch* commands. The heuristic signals that there are additional keys available for each bundle that help determine which subset of bundles the client should download. The only value currently understood is *creationToken*.

bundle.<id>.*

The *bundle.<id>.** keys are used to describe a single item in the bundle list, grouped under *<id>* for identification purposes.

bundle.<id>.uri

This string value defines the URI by which Git can reach the contents of this *<id>*. This URI may be a bundle file or another bundle list.

checkout.defaultRemote

When you run *git checkout <something>* or *git switch <something>* and only have one remote, it may implicitly fall back on checking out and tracking e.g. *origin/<something>*. This stops working as soon as you have more than one remote with a *<something>* reference. This setting allows for setting the name of a preferred remote that should always win when it comes to disambiguation. The typical use-case is to set this to *origin*.

Currently this is used by Section G.3.140, “git-switch(1)” and Section G.3.19, “git-checkout(1)” when *git checkout <something>* or *git switch <something>* will checkout the *<something>* branch on another remote, and by Section G.3.159, “git-worktree(1)” when *git worktree add* refers to a remote branch. This setting might be used for other checkout-like commands or functionality in the future.

checkout.guess

Provides the default value for the *--guess* or *--no-guess* option in *git checkout* and *git switch*. See Section G.3.140, “git-switch(1)” and Section G.3.19, “git-checkout(1)”.

checkout.workers

The number of parallel workers to use when updating the working tree. The default is one, i.e. sequential execution. If set to a value less than one, Git will use as many workers as the number of logical cores available. This setting and *checkout.thresholdForParallelism* affect all commands that perform checkout. E.g. checkout, clone, reset, sparse-checkout, etc.

Note: Parallel checkout usually delivers better performance for repositories located on SSDs or over NFS. For repositories on spinning disks and/or machines with a small number of cores, the default sequential checkout often performs better. The size and compression level of a repository might also influence how well the parallel version performs.

checkout.thresholdForParallelism

When running parallel checkout with a small number of files, the cost of subprocess spawning and inter-process communication might outweigh the parallelization gains. This setting allows you to define the minimum number of files for which parallel checkout should be attempted. The default is 100.

clean.requireForce

A boolean to make git-clean refuse to delete files unless -f is given. Defaults to true.

clone.defaultRemoteName

The name of the remote to create when cloning a repository. Defaults to *origin*. It can be overridden by passing the *--origin* command-line option to Section G.3.24, "git-clone(1)".

clone.rejectShallow

Reject cloning a repository if it is a shallow one; this can be overridden by passing the *--reject-shallow* option on the command line. See Section G.3.24, "git-clone(1)".

clone.filterSubmodules

If a partial clone filter is provided (see *--filter* in Section G.3.120, "git-rev-list(1)") and *--recurse-submodules* is used, also apply the filter to submodules.

color.advice

A boolean to enable/disable color in hints (e.g. when a push failed, see *advice.** for a list). May be set to *always*, *false* (or *never*) or *auto* (or *true*), in which case colors are used only when the error output goes to a terminal. If unset, then the value of *color.ui* is used (*auto* by default).

color.advice.hint

Use customized color for hints.

color.blame.highlightRecent

Specify the line annotation color for *git blame --color-by-age* depending upon the age of the line.

This setting should be set to a comma-separated list of color and date settings, starting and ending with a color, the dates should be set from oldest to newest. The metadata will be colored with the specified colors if the line was introduced before the given timestamp, overwriting older timestamped colors.

Instead of an absolute timestamp relative timestamps work as well, e.g. *2.weeks.ago* is valid to address anything older than 2 weeks.

It defaults to *blue,12 month ago,white,1 month ago,red*, which colors everything older than one year blue, recent changes between one month and one year old are kept white, and lines introduced within the last month are colored red.

color.blame.repeatedLines

Use the specified color to colorize line annotations for *git blame --color-lines*, if they come from the same commit as the preceding line. Defaults to cyan.

color.branch

A boolean to enable/disable color in the output of Section G.3.10, "git-branch(1)". May be set to *always*, *false* (or *never*) or *auto* (or *true*), in which case colors are used only when the output is to a terminal. If unset, then the value of *color.ui* is used (*auto* by default).

color.branch.<slot>

Use customized color for branch coloration. *<slot>* is one of *current* (the current branch), *local* (a local branch), *remote* (a remote-tracking branch in refs/remotes/), *upstream* (upstream tracking branch), *plain* (other refs).

color.diff

Whether to use ANSI escape sequences to add color to patches. If this is set to *always*, Section G.3.44, "git-diff(1)", Section G.3.74, "git-log(1)", and Section G.3.134, "git-show(1)" will use color for all patches. If it

is set to *true* or *auto*, those commands will only use color when output is to the terminal. If unset, then the value of *color.ui* is used (*auto* by default).

This does not affect Section G.3.54, "git-format-patch(1)" or the *git-diff-** plumbing commands. Can be overridden on the command line with the *--color[=<when>]* option.

color.diff.<slot>

Use customized color for diff colorization. *<slot>* specifies which part of the patch to use the specified color, and is one of *context* (context text - *plain* is a historical synonym), *meta* (metainformation), *frag* (hunk header), *func* (function in hunk header), *old* (removed lines), *new* (added lines), *commit* (commit headers), *whitespace* (highlighting whitespace errors), *oldMoved* (deleted lines), *newMoved* (added lines), *oldMovedDimmed*, *oldMovedAlternative*, *oldMovedAlternativeDimmed*, *newMovedDimmed*, *newMovedAlternative* *newMovedAlternativeDimmed* (See the *<mode>* setting of *--color-moved* in Section G.3.44, "git-diff(1)" for details), *contextDimmed*, *oldDimmed*, *newDimmed*, *contextBold*, *oldBold*, and *newBold* (see Section G.3.105, "git-range-diff(1)" for details).

color.decorate.<slot>

Use customized color for *git log --decorate* output. *<slot>* is one of *branch*, *remoteBranch*, *tag*, *stash* or *HEAD* for local branches, remote-tracking branches, tags, stash and HEAD, respectively and *grafted* for grafted commits.

color.grep

When set to *always*, always highlight matches. When *false* (or *never*), never. When set to *true* or *auto*, use color only when the output is written to the terminal. If unset, then the value of *color.ui* is used (*auto* by default).

color.grep.<slot>

Use customized color for grep colorization. *<slot>* specifies which part of the line to use the specified color, and is one of

context

non-matching text in context lines (when using *-A*, *-B*, or *-C*)

filename

filename prefix (when not using *-h*)

function

function name lines (when using *-p*)

lineNumber

line number prefix (when using *-n*)

column

column number prefix (when using *--column*)

match

matching text (same as setting *matchContext* and *matchSelected*)

matchContext

matching text in context lines

matchSelected

matching text in selected lines. Also, used to customize the following Section G.3.74, “git-log(1)” subcommands: *--grep*, *--author*, and *--committer*.

selected

non-matching text in selected lines. Also, used to customize the following Section G.3.74, “git-log(1)” subcommands: *--grep*, *--author* and *--committer*.

separator

separators between fields on a line (*:*, -, and =) and between hunks (--)

color.interactive

When set to *always*, always use colors for interactive prompts and displays (such as those used by "git-add --interactive" and "git-clean --interactive"). When false (or *never*), never. When set to *true* or *auto*, use colors only when the output is to the terminal. If unset, then the value of *color.ui* is used (*auto* by default).

color.interactive.<slot>

Use customized color for *git add --interactive* and *git clean --interactive* output. <*slot*> may be *prompt*, *header*, *help* or *error*, for four distinct types of normal output from interactive commands.

color.pager

A boolean to specify whether *auto* color modes should colorize output going to the pager. Defaults to true; set this to false if your pager does not understand ANSI color codes.

color.push

A boolean to enable/disable color in push errors. May be set to *always*, *false* (or *never*) or *auto* (or *true*), in which case colors are used only when the error output goes to a terminal. If unset, then the value of *color.ui* is used (*auto* by default).

color.push.error

Use customized color for push errors.

color.remote

If set, keywords at the start of the line are highlighted. The keywords are "error", "warning", "hint" and "success", and are matched case-insensitively. May be set to *always*, *false* (or *never*) or *auto* (or *true*). If unset, then the value of *color.ui* is used (*auto* by default).

color.remote.<slot>

Use customized color for each remote keyword. <*slot*> may be *hint*, *warning*, *success* or *error* which match the corresponding keyword.

color.showBranch

A boolean to enable/disable color in the output of Section G.3.131, “git-show-branch(1)”. May be set to *always*, *false* (or *never*) or *auto* (or *true*), in which case colors are used only when the output is to a terminal. If unset, then the value of *color.ui* is used (*auto* by default).

color.status

A boolean to enable/disable color in the output of Section G.3.138, “git-status(1)”. May be set to *always*, *false* (or *never*) or *auto* (or *true*), in which case colors are used only when the output is to a terminal. If unset, then the value of *color.ui* is used (*auto* by default).

color.status.<slot>

Use customized color for status colorization. *<slot>* is one of *header* (the header text of the status message), *added* or *updated* (files which are added but not committed), *changed* (files which are changed but not added in the index), *untracked* (files which are not tracked by Git), *branch* (the current branch), *nobranch* (the color the *no branch* warning is shown in, defaulting to red), *localBranch* or *remoteBranch* (the local and remote branch names, respectively, when branch and tracking information is displayed in the status short-format), or *unmerged* (files which have unmerged changes).

color.transport

A boolean to enable/disable color when pushes are rejected. May be set to *always*, *false* (or *never*) or *auto* (or *true*), in which case colors are used only when the error output goes to a terminal. If unset, then the value of *color.ui* is used (*auto* by default).

color.transport.rejected

Use customized color when a push was rejected.

color.ui

This variable determines the default value for variables such as *color.diff* and *color.grep* that control the use of color per command family. Its scope will expand as more commands learn configuration to set a default for the *--color* option. Set it to *false* or *never* if you prefer Git commands not to use color unless enabled explicitly with some other configuration or the *--color* option. Set it to *always* if you want all output not intended for machine consumption to use color, to *true* or *auto* (this is the default since Git 1.8.4) if you want such output to use color when written to the terminal.

column.ui

Specify whether supported commands should output in columns. This variable consists of a list of tokens separated by spaces or commas:

These options control when the feature should be enabled (defaults to *never*):

always

always show in columns

never

never show in columns

auto

show in columns if the output is to the terminal

These options control layout (defaults to *column*). Setting any of these implies *always* if none of *always*, *never*, or *auto* are specified.

column

fill columns before rows

row

fill rows before columns

plain

show in one column

Finally, these options can be combined with a layout option (defaults to *nodense*):

dense

make unequal size columns to utilize more space

nodense

make equal size columns

column.branch

Specify whether to output branch listing in *git branch* in columns. See *column.ui* for details.

column.clean

Specify the layout when listing items in *git clean -i*, which always shows files and directories in columns. See *column.ui* for details.

column.status

Specify whether to output untracked files in *git status* in columns. See *column.ui* for details.

column.tag

Specify whether to output tag listings in *git tag* in columns. See *column.ui* for details.

commit.cleanup

This setting overrides the default of the *--cleanup* option in *git commit*. See Section G.3.28, “git-commit(1)” for details. Changing the default can be useful when you always want to keep lines that begin with the comment character # in your log message, in which case you would do *git config commit.cleanup whitespace* (note that you will have to remove the help lines that begin with # in the commit log template yourself, if you do this).

commit.gpgSign

A boolean to specify whether all commits should be GPG signed. Use of this option when doing operations such as rebase can result in a large number of commits being signed. It may be convenient to use an agent to avoid typing your GPG passphrase several times.

commit.status

A boolean to enable/disable inclusion of status information in the commit message template when using an editor to prepare the commit message. Defaults to true.

commit.template

Specify the pathname of a file to use as the template for new commit messages.

commit.verbose

A boolean or int to specify the level of verbosity with *git commit*. See Section G.3.28, “git-commit(1)”.

commitGraph.generationVersion

Specifies the type of generation number version to use when writing or reading the commit-graph file. If version 1 is specified, then the corrected commit dates will not be written or read. Defaults to 2.

commitGraph.maxNewFilters

Specifies the default value for the *--max-new-filters* option of *git commit-graph write* (c.f., Section G.3.26, “git-commit-graph(1)”).

commitGraph.readChangedPaths

If true, then git will use the changed-path Bloom filters in the commit-graph file (if it exists, and they are present). Defaults to true. See Section G.3.26, “git-commit-graph(1)” for more information.

completion.commands

This is only used by git-completion.bash to add or remove commands from the list of completed commands. Normally only porcelain commands and a few select others are completed. You can add more commands, separated by space, in this variable. Prefixing the command with - will remove it from the existing list.

core.fileMode

Tells Git if the executable bit of files in the working tree is to be honored.

Some filesystems lose the executable bit when a file that is marked as executable is checked out, or checks out a non-executable file with executable bit on. Section G.3.24, “git-clone(1)” or Section G.3.71, “git-init(1)” probe the filesystem to see if it handles the executable bit correctly and this variable is automatically set as necessary.

A repository, however, may be on a filesystem that handles the filemode correctly, and this variable is set to *true* when created, but later may be made accessible from another environment that loses the filemode (e.g. exporting ext4 via CIFS mount, visiting a Cygwin created repository with Git for Windows or Eclipse). In such a case it may be necessary to set this variable to *false*. See Section G.3.147, “git-update-index(1)”.

The default is true (when core.filemode is not specified in the config file).

core.hideDotFiles

(Windows-only) If true, mark newly-created directories and files whose name starts with a dot as hidden. If *dotGitOnly*, only the *.git/* directory is hidden, but no other files starting with a dot. The default mode is *dotGitOnly*.

core.ignoreCase

Internal variable which enables various workarounds to enable Git to work better on filesystems that are not case sensitive, like APFS, HFS+, FAT, NTFS, etc. For example, if a directory listing finds "makefile" when Git expects "Makefile", Git will assume it is really the same file, and continue to remember it as "Makefile".

The default is false, except Section G.3.24, “git-clone(1)” or Section G.3.71, “git-init(1)” will probe and set core.ignoreCase true if appropriate when the repository is created.

Git relies on the proper configuration of this variable for your operating and file system. Modifying this value may result in unexpected behavior.

core.precomposeUnicode

This option is only used by Mac OS implementation of Git. When core.precomposeUnicode=true, Git reverts the unicode decomposition of filenames done by Mac OS. This is useful when sharing a repository between Mac OS and Linux or Windows. (Git for Windows 1.7.10 or higher is needed, or Git under cygwin 1.7). When false, file names are handled fully transparent by Git, which is backward compatible with older versions of Git.

core.protectHFS

If set to true, do not allow checkout of paths that would be considered equivalent to *.git* on an HFS+ filesystem. Defaults to *true* on Mac OS, and *false* elsewhere.

core.protectNTFS

If set to true, do not allow checkout of paths that would cause problems with the NTFS filesystem, e.g. conflict with 8.3 "short" names. Defaults to *true* on Windows, and *false* elsewhere.

core.fsmonitor

If set to true, enable the built-in file system monitor daemon for this working directory (Section G.3.57, “git-fsmonitor--daemon(1)”).

Like hook-based file system monitors, the built-in file system monitor can speed up Git commands that need to refresh the Git index (e.g. *git status*) in a working directory with many files. The built-in monitor eliminates the need to install and maintain an external third-party tool.

The built-in file system monitor is currently available only on a limited set of supported platforms. Currently, this includes Windows and MacOS.

```
Otherwise, this variable contains the pathname of the "fsmonitor"
hook command.
```

This hook command is used to identify all files that may have changed since the requested date/time. This information is used to speed up git by avoiding unnecessary scanning of files that have not changed.

See the "fsmonitor-watchman" section of Section G.4.7, “githooks(5)”.

Note that if you concurrently use multiple versions of Git, such as one version on the command line and another version in an IDE tool, that the definition of *core.fsmonitor* was extended to allow boolean values in addition to hook pathnames. Git versions 2.35.1 and prior will not understand the boolean values and will consider the "true" or "false" values as hook pathnames to be invoked. Git versions 2.26 thru 2.35.1 default to hook protocol V2 and will fall back to no fsmonitor (full scan). Git versions prior to 2.26 default to hook protocol V1 and will silently assume there were no changes to report (no scan), so status commands may report incomplete results. For this reason, it is best to upgrade all of your Git versions before using the built-in file system monitor.

core.fsmonitorHookVersion

Sets the protocol version to be used when invoking the "fsmonitor" hook.

There are currently versions 1 and 2. When this is not set, version 2 will be tried first and if it fails then version 1 will be tried. Version 1 uses a timestamp as input to determine which files have changes since that time but some monitors like Watchman have race conditions when used with a timestamp. Version 2 uses an opaque string so that the monitor can return something that can be used to determine what files have changed without race conditions.

core.trustctime

If false, the ctime differences between the index and the working tree are ignored; useful when the inode change time is regularly modified by something outside Git (file system crawlers and some backup systems). See Section G.3.147, “git-update-index(1)”. True by default.

core.splitIndex

If true, the split-index feature of the index will be used. See Section G.3.147, “git-update-index(1)”. False by default.

core.untrackedCache

Determines what to do about the untracked cache feature of the index. It will be kept, if this variable is unset or set to *keep*. It will automatically be added if set to *true*. And it will automatically be removed, if set to *false*. Before setting it to *true*, you should check that mtime is working properly on your system. See Section G.3.147, “git-update-index(1)”. *keep* by default, unless *feature.manyFiles* is enabled which sets this setting to *true* by default.

core.checkStat

When missing or is set to *default*, many fields in the stat structure are checked to detect if a file has been modified since Git looked at it. When this configuration variable is set to *minimal*, sub-second part of mtime

and ctime, the uid and gid of the owner of the file, the inode number (and the device number, if Git was compiled to use it), are excluded from the check among these fields, leaving only the whole-second part of mtime (and ctime, if *core.trustCtime* is set) and the filesize to be checked.

There are implementations of Git that do not leave usable values in some fields (e.g. JGit); by excluding these fields from the comparison, the *minimal* mode may help interoperability when the same repository is used by these other systems at the same time.

core.quotePath

Commands that output paths (e.g. *ls-files*, *diff*), will quote "unusual" characters in the pathname by enclosing the pathname in double-quotes and escaping those characters with backslashes in the same way C escapes control characters (e.g. *\t* for TAB, *\n* for LF, \\ for backslash) or bytes with values larger than 0x80 (e.g. octal *\302\265* for "micro" in UTF-8). If this variable is set to false, bytes higher than 0x80 are not considered "unusual" any more. Double-quotes, backslash and control characters are always escaped regardless of the setting of this variable. A simple space character is not considered "unusual". Many commands can output pathnames completely verbatim using the *-z* option. The default value is true.

core.eol

Sets the line ending type to use in the working directory for files that are marked as text (either by having the *text* attribute set, or by having *text=auto* and Git auto-detecting the contents as text). Alternatives are *lf*, *crlf* and *native*, which uses the platform's native line ending. The default value is *native*. See Section G.4.2, “gitattributes(5)” for more information on end-of-line conversion. Note that this value is ignored if *core.autocrlf* is set to *true* or *input*.

core.safecrlf

If true, makes Git check if converting *CRLF* is reversible when end-of-line conversion is active. Git will verify if a command modifies a file in the work tree either directly or indirectly. For example, committing a file followed by checking out the same file should yield the original file in the work tree. If this is not the case for the current setting of *core.autocrlf*, Git will reject the file. The variable can be set to "warn", in which case Git will only warn about an irreversible conversion but continue the operation.

CRLF conversion bears a slight chance of corrupting data. When it is enabled, Git will convert CRLF to LF during commit and LF to CRLF during checkout. A file that contains a mixture of LF and CRLF before the commit cannot be recreated by Git. For text files this is the right thing to do: it corrects line endings such that we have only LF line endings in the repository. But for binary files that are accidentally classified as text the conversion can corrupt data.

If you recognize such corruption early you can easily fix it by setting the conversion type explicitly in .gitattributes. Right after committing you still have the original file in your work tree and this file is not yet corrupted. You can explicitly tell Git that this file is binary and Git will handle the file appropriately.

Unfortunately, the desired effect of cleaning up text files with mixed line endings and the undesired effect of corrupting binary files cannot be distinguished. In both cases CRLFs are removed in an irreversible way. For text files this is the right thing to do because CRLFs are line endings, while for binary files converting CRLFs corrupts data.

Note, this safety check does not mean that a checkout will generate a file identical to the original file for a different setting of *core.eol* and *core.autocrlf*, but only for the current one. For example, a text file with *LF* would be accepted with *core.eol=lf* and could later be checked out with *core.eol=crlf*, in which case the resulting file would contain *CRLF*, although the original file contained *LF*. However, in both work trees the line endings would be consistent, that is either all *LF* or all *CRLF*, but never mixed. A file with mixed line endings would be reported by the *core.safecrlf* mechanism.

core.autocrlf

Setting this variable to "true" is the same as setting the *text* attribute to "auto" on all files and core.eol to "crlf". Set to true if you want to have *CRLF* line endings in your working directory and the repository has LF line endings. This variable can be set to *input*, in which case no output conversion is performed.

core.checkRoundtripEncoding

A comma and/or whitespace separated list of encodings that Git performs UTF-8 round trip checks on if they are used in an *working-tree-encoding* attribute (see Section G.4.2, “gitattributes(5)”). The default value is *SHIFT-JIS*.

core.symlinks

If false, symbolic links are checked out as small plain files that contain the link text. Section G.3.147, “git-update-index(1)” and Section G.3.2, “git-add(1)” will not change the recorded type to regular file. Useful on filesystems like FAT that do not support symbolic links.

The default is true, except Section G.3.24, “git-clone(1)” or Section G.3.71, “git-init(1)” will probe and set core.symlinks false if appropriate when the repository is created.

core.gitProxy

A "proxy command" to execute (as *command host port*) instead of establishing direct connection to the remote server when using the Git protocol for fetching. If the variable value is in the "COMMAND for DOMAIN" format, the command is applied only on hostnames ending with the specified domain string. This variable may be set multiple times and is matched in the given order; the first match wins.

Can be overridden by the *GIT_PROXY_COMMAND* environment variable (which always applies universally, without the special "for" handling).

The special string *none* can be used as the proxy command to specify that no proxy be used for a given domain pattern. This is useful for excluding servers inside a firewall from proxy use, while defaulting to a common proxy for external domains.

core.sshCommand

If this variable is set, *git fetch* and *git push* will use the specified command instead of *ssh* when they need to connect to a remote system. The command is in the same form as the *GIT_SSH_COMMAND* environment variable and is overridden when the environment variable is set.

core.ignoreStat

If true, Git will avoid using lstat() calls to detect if files have changed by setting the "assume-unchanged" bit for those tracked files which it has updated identically in both the index and working tree.

When files are modified outside of Git, the user will need to stage the modified files explicitly (e.g. see *Examples* section in Section G.3.147, “git-update-index(1)”). Git will not normally detect changes to those files.

This is useful on systems where lstat() calls are very slow, such as CIFS/Microsoft Windows.

False by default.

core.preferSymlinkRefs

Instead of the default "symref" format for HEAD and other symbolic reference files, use symbolic links. This is sometimes needed to work with old scripts that expect HEAD to be a symbolic link.

core.alternateRefsCommand

When advertising tips of available history from an alternate, use the shell to execute the specified command instead of Section G.3.52, “git-for-each-ref(1)”. The first argument is the absolute path of the alternate. Output must contain one hex object id per line (i.e., the same as produced by *git for-each-ref --format='%(objectname)'*).

Note that you cannot generally put *git for-each-ref* directly into the config value, as it does not take a repository path as an argument (but you can wrap the command above in a shell script).

core.alternateRefsPrefixes

When listing references from an alternate, list only references that begin with the given prefix. Prefixes match as if they were given as arguments to Section G.3.52, “git-for-each-ref(1)”. To list multiple prefixes, separate them with whitespace. If *core.alternateRefsCommand* is set, setting *core.alternateRefsPrefixes* has no effect.

core.bare

If true this repository is assumed to be *bare* and has no working directory associated with it. If this is the case a number of commands that require a working directory will be disabled, such as Section G.3.2, “git-add(1)” or Section G.3.86, “git-merge(1)”.

This setting is automatically guessed by Section G.3.24, “git-clone(1)” or Section G.3.71, “git-init(1)” when the repository was created. By default a repository that ends in "/.git" is assumed to be not bare (bare = false), while all other repositories are assumed to be bare (bare = true).

core.worktree

Set the path to the root of the working tree. If *GIT_COMMON_DIR* environment variable is set, core.worktree is ignored and not used for determining the root of working tree. This can be overridden by the *GIT_WORK_TREE* environment variable and the *--work-tree* command-line option. The value can be an absolute path or relative to the path to the .git directory, which is either specified by --git-dir or GIT_DIR, or automatically discovered. If --git-dir or GIT_DIR is specified but none of --work-tree, GIT_WORK_TREE and core.worktree is specified, the current working directory is regarded as the top level of your working tree.

Note that this variable is honored even when set in a configuration file in a ".git" subdirectory of a directory and its value differs from the latter directory (e.g. "/path/to/.git/config" has core.worktree set to "/different/path"), which is most likely a misconfiguration. Running Git commands in the "/path/to" directory will still use "/different/path" as the root of the work tree and can cause confusion unless you know what you are doing (e.g. you are creating a read-only snapshot of the same index to a location different from the repository's usual working tree).

core.logAllRefUpdates

Enable the reflog. Updates to a ref <ref> is logged to the file "*$GIT_DIR/logs/<ref>*", by appending the new and old SHA-1, the date/time and the reason of the update, but only when the file exists. If this configuration variable is set to *true*, missing "*$GIT_DIR/logs/<ref>*" file is automatically created for branch heads (i.e. under *refs/heads/*), remote refs (i.e. under *refs/remotes/*), note refs (i.e. under *refs/notes/*), and the symbolic ref *HEAD*. If it is set to *always*, then a missing reflog is automatically created for any ref under *refs/*.

This information can be used to determine what commit was the tip of a branch "2 days ago".

This value is true by default in a repository that has a working directory associated with it, and false by default in a bare repository.

core.repositoryFormatVersion

Internal variable identifying the repository format and layout version.

core.sharedRepository

When *group* (or *true*), the repository is made shareable between several users in a group (making sure all the files and objects are group-writable). When *all* (or *world* or *everybody*), the repository will be readable by all users, additionally to being group-shareable. When *umask* (or *false*), Git will use permissions reported by umask(2). When *0xxx*, where *0xxx* is an octal number, files in the repository will have this mode value. *0xxx* will override user's umask value (whereas the other options will only override requested parts of the user's umask value). Examples: *0660* will make the repo read/write-able for the owner and group, but inaccessible to others (equivalent to *group* unless umask is e.g. *0022*). *0640* is a repository that is group-readable but not group-writable. See Section G.3.71, “git-init(1)”. False by default.

core.warnAmbiguousRefs

If true, Git will warn you if the ref name you passed it is ambiguous and might match multiple refs in the repository. True by default.

core.compression

An integer -1..9, indicating a default compression level. -1 is the zlib default. 0 means no compression, and 1..9 are various speed/size tradeoffs, 9 being slowest. If set, this provides a default to other compression variables, such as *core.looseCompression* and *pack.compression*.

core.looseCompression

An integer -1..9, indicating the compression level for objects that are not in a pack file. -1 is the zlib default. 0 means no compression, and 1..9 are various speed/size tradeoffs, 9 being slowest. If not set, defaults to core.compression. If that is not set, defaults to 1 (best speed).

core.packedGitWindowSize

Number of bytes of a pack file to map into memory in a single mapping operation. Larger window sizes may allow your system to process a smaller number of large pack files more quickly. Smaller window sizes will negatively affect performance due to increased calls to the operating system's memory manager, but may improve performance when accessing a large number of large pack files.

Default is 1 MiB if NO_MMAP was set at compile time, otherwise 32 MiB on 32 bit platforms and 1 GiB on 64 bit platforms. This should be reasonable for all users/operating systems. You probably do not need to adjust this value.

Common unit suffixes of *k*, *m*, or *g* are supported.

core.packedGitLimit

Maximum number of bytes to map simultaneously into memory from pack files. If Git needs to access more than this many bytes at once to complete an operation it will unmap existing regions to reclaim virtual address space within the process.

Default is 256 MiB on 32 bit platforms and 32 TiB (effectively unlimited) on 64 bit platforms. This should be reasonable for all users/operating systems, except on the largest projects. You probably do not need to adjust this value.

Common unit suffixes of *k*, *m*, or *g* are supported.

core.deltaBaseCacheLimit

Maximum number of bytes per thread to reserve for caching base objects that may be referenced by multiple deltified objects. By storing the entire decompressed base objects in a cache Git is able to avoid unpacking and decompressing frequently used base objects multiple times.

Default is 96 MiB on all platforms. This should be reasonable for all users/operating systems, except on the largest projects. You probably do not need to adjust this value.

Common unit suffixes of *k*, *m*, or *g* are supported.

core.bigFileThreshold

The size of files considered "big", which as discussed below changes the behavior of numerous git commands, as well as how such files are stored within the repository. The default is 512 MiB. Common unit suffixes of *k*, *m*, or *g* are supported.

Files above the configured limit will be:

- Stored deflated in packfiles, without attempting delta compression.

The default limit is primarily set with this use-case in mind. With it, most projects will have their source code and other text files delta compressed, but not larger binary media files.

Storing large files without delta compression avoids excessive memory usage, at the slight expense of increased disk usage.

- Will be treated as if they were labeled "binary" (see Section G.4.2, “gitattributes(5)”). e.g. Section G.3.74, “git-log(1)” and Section G.3.44, “git-diff(1)” will not compute diffs for files above this limit.
- Will generally be streamed when written, which avoids excessive memory usage, at the cost of some fixed overhead. Commands that make use of this include Section G.3.7, “git-archive(1)”, Section G.3.47, “git-fast-import(1)”, Section G.3.69, “git-index-pack(1)”, Section G.3.146, “git-unpack-objects(1)” and Section G.3.56, “git-fsck(1)”.

core.excludesFile

Specifies the pathname to the file that contains patterns to describe paths that are not meant to be tracked, in addition to *.gitignore* (per-directory) and *.git/info/exclude*. Defaults to *$XDG_CONFIG_HOME/git/ignore*. If *$XDG_CONFIG_HOME* is either not set or empty, *$HOME/.config/git/ignore* is used instead. See Section G.4.5, “gitignore(5)”.

core.askPass

Some commands (e.g. svn and http interfaces) that interactively ask for a password can be told to use an external program given via the value of this variable. Can be overridden by the *GIT_ASKPASS* environment variable. If not set, fall back to the value of the *SSH_ASKPASS* environment variable or, failing that, a simple password prompt. The external program shall be given a suitable prompt as command-line argument and write the password on its STDOUT.

core.attributesFile

In addition to *.gitattributes* (per-directory) and *.git/info/attributes*, Git looks into this file for attributes (see Section G.4.2, “gitattributes(5)”). Path expansions are made the same way as for *core.excludesFile*. Its default value is *$XDG_CONFIG_HOME/git/attributes*. If *$XDG_CONFIG_HOME* is either not set or empty, *$HOME/.config/git/attributes* is used instead.

core.hooksPath

By default Git will look for your hooks in the *$GIT_DIR/hooks* directory. Set this to different path, e.g. */etc/git/hooks*, and Git will try to find your hooks in that directory, e.g. */etc/git/hooks/pre-receive* instead of in *$GIT_DIR/hooks/pre-receive*.

The path can be either absolute or relative. A relative path is taken as relative to the directory where the hooks are run (see the "DESCRIPTION" section of Section G.4.7, “githooks(5)”).

This configuration variable is useful in cases where you'd like to centrally configure your Git hooks instead of configuring them on a per-repository basis, or as a more flexible and centralized alternative to having an *init.templateDir* where you've changed default hooks.

core.editor

Commands such as *commit* and *tag* that let you edit messages by launching an editor use the value of this variable when it is set, and the environment variable *GIT_EDITOR* is not set. See Section G.3.152, “git-var(1)”.

core.commentChar , core.commentString

Commands such as *commit* and *tag* that let you edit messages consider a line that begins with this character commented, and removes them after the editor returns (default #).

If set to "auto", *git-commit* would select a character that is not the beginning character of any line in existing commit messages.

Note that these two variables are aliases of each other, and in modern versions of Git you are free to use a string (e.g., *//* or *###*) with *commentChar*. Versions of Git prior to v2.45.0 will ignore *commentString* but will reject a value of *commentChar* that consists of more than a single ASCII byte. If you plan to use your config with older and newer versions of Git, you may want to specify both:

```
[core]
# single character for older versions
commentChar = "#"
# string for newer versions (which will override commentChar
# because it comes later in the file)
commentString = "//"
```

core.filesRefLockTimeout

The length of time, in milliseconds, to retry when trying to lock an individual reference. Value 0 means not to retry at all; -1 means to try indefinitely. Default is 100 (i.e., retry for 100ms).

core.packedRefsTimeout

The length of time, in milliseconds, to retry when trying to lock the *packed-refs* file. Value 0 means not to retry at all; -1 means to try indefinitely. Default is 1000 (i.e., retry for 1 second).

core.pager

Text viewer for use by Git commands (e.g., *less*). The value is meant to be interpreted by the shell. The order of preference is the *$GIT_PAGER* environment variable, then *core.pager* configuration, then *$PAGER*, and then the default chosen at compile time (usually *less*).

When the *LESS* environment variable is unset, Git sets it to *FRX* (if *LESS* environment variable is set, Git does not change it at all). If you want to selectively override Git's default setting for *LESS*, you can set *core.pager* to e.g. *less -S*. This will be passed to the shell by Git, which will translate the final command to *LESS=FRX less -S*. The environment does not set the *S* option but the command line does, instructing less to truncate long lines. Similarly, setting *core.pager* to *less -+F* will deactivate the *F* option specified by the environment from the command-line, deactivating the "quit if one screen" behavior of *less*. One can specifically activate some flags for particular commands: for example, setting *pager.blame* to *less -S* enables line truncation only for *git blame*.

Likewise, when the *LV* environment variable is unset, Git sets it to *-c*. You can override this setting by exporting *LV* with another value or setting *core.pager* to *lv +c*.

core.whitespace

A comma separated list of common whitespace problems to notice. *git diff* will use *color.diff.whitespace* to highlight them, and *git apply --whitespace=error* will consider them as errors. You can prefix - to disable any of them (e.g. *-trailing-space*):

- *blank-at-eol* treats trailing whitespaces at the end of the line as an error (enabled by default).
- *space-before-tab* treats a space character that appears immediately before a tab character in the initial indent part of the line as an error (enabled by default).
- *indent-with-non-tab* treats a line that is indented with space characters instead of the equivalent tabs as an error (not enabled by default).
- *tab-in-indent* treats a tab character in the initial indent part of the line as an error (not enabled by default).
- *blank-at-eof* treats blank lines added at the end of file as an error (enabled by default).
- *trailing-space* is a short-hand to cover both *blank-at-eol* and *blank-at-eof*.

- *cr-at-eol* treats a carriage-return at the end of line as part of the line terminator, i.e. with it, *trailing-space* does not trigger if the character before such a carriage-return is not a whitespace (not enabled by default).
- *tabwidth=<n>* tells how many character positions a tab occupies; this is relevant for *indent-with-non-tab* and when Git fixes *tab-in-indent* errors. The default tab width is 8. Allowed values are 1 to 63.

core.fsync

A comma-separated list of components of the repository that should be hardened via the core.fsyncMethod when created or modified. You can disable hardening of any component by prefixing it with a -. Items that are not hardened may be lost in the event of an unclean system shutdown. Unless you have special requirements, it is recommended that you leave this option empty or pick one of *committed*, *added*, or *all*.

When this configuration is encountered, the set of components starts with the platform default value, disabled components are removed, and additional components are added. *none* resets the state so that the platform default is ignored.

The empty string resets the fsync configuration to the platform default. The default on most platforms is equivalent to *core.fsync=committed,-loose-object*, which has good performance, but risks losing recent work in the event of an unclean system shutdown.

- *none* clears the set of fsynced components.
- *loose-object* hardens objects added to the repo in loose-object form.
- *pack* hardens objects added to the repo in packfile form.
- *pack-metadata* hardens packfile bitmaps and indexes.
- *commit-graph* hardens the commit-graph file.
- *index* hardens the index when it is modified.
- *objects* is an aggregate option that is equivalent to *loose-object,pack*.
- *reference* hardens references modified in the repo.
- *derived-metadata* is an aggregate option that is equivalent to *pack-metadata,commit-graph*.
- *committed* is an aggregate option that is currently equivalent to *objects*. This mode sacrifices some performance to ensure that work that is committed to the repository with *git commit* or similar commands is hardened.
- *added* is an aggregate option that is currently equivalent to *committed,index*. This mode sacrifices additional performance to ensure that the results of commands like *git add* and similar operations are hardened.
- *all* is an aggregate option that syncs all individual components above.

core.fsyncMethod

A value indicating the strategy Git will use to harden repository data using fsync and related primitives.

- *fsync* uses the fsync() system call or platform equivalents.
- *writeout-only* issues pagecache writeback requests, but depending on the filesystem and storage hardware, data added to the repository may not be durable in the event of a system crash. This is the default mode on macOS.
- *batch* enables a mode that uses writeout-only flushes to stage multiple updates in the disk writeback cache and then does a single full fsync of a dummy file to trigger the disk cache flush at the end of the operation.

  Currently *batch* mode only applies to loose-object files. Other repository data is made durable as if *fsync* was specified. This mode is expected to be as safe as *fsync* on macOS for repos stored on HFS+ or APFS filesystems and on Windows for repos stored on NTFS or ReFS filesystems.

core.fsyncObjectFiles

This boolean will enable *fsync()* when writing object files. This setting is deprecated. Use core.fsync instead.

This setting affects data added to the Git repository in loose-object form. When set to true, Git will issue an fsync or similar system call to flush caches so that loose-objects remain consistent in the face of a unclean system shutdown.

core.preloadIndex

Enable parallel index preload for operations like *git diff*

This can speed up operations like *git diff* and *git status* especially on filesystems like NFS that have weak caching semantics and thus relatively high IO latencies. When enabled, Git will do the index comparison to the filesystem data in parallel, allowing overlapping IO's. Defaults to true.

core.unsetenvvars

Windows-only: comma-separated list of environment variables' names that need to be unset before spawning any other process. Defaults to *PERL5LIB* to account for the fact that Git for Windows insists on using its own Perl interpreter.

core.restrictinheritedhandles

Windows-only: override whether spawned processes inherit only standard file handles (*stdin*, *stdout* and *stderr*) or all handles. Can be *auto*, *true* or *false*. Defaults to *auto*, which means *true* on Windows 7 and later, and *false* on older Windows versions.

core.createObject

You can set this to *link*, in which case a hardlink followed by a delete of the source are used to make sure that object creation will not overwrite existing objects.

On some file system/operating system combinations, this is unreliable. Set this config setting to *rename* there; however, this will remove the check that makes sure that existing object files will not get overwritten.

core.notesRef

When showing commit messages, also show notes which are stored in the given ref. The ref must be fully qualified. If the given ref does not exist, it is not an error but means that no notes should be printed.

This setting defaults to "refs/notes/commits", and it can be overridden by the *GIT_NOTES_REF* environment variable. See Section G.3.94, “git-notes(1)”.

core.commitGraph

If true, then git will read the commit-graph file (if it exists) to parse the graph structure of commits. Defaults to true. See Section G.3.26, “git-commit-graph(1)” for more information.

core.useReplaceRefs

If set to *false*, behave as if the *--no-replace-objects* option was given on the command line. See Section G.3.1, “git(1)” and Section G.3.114, “git-replace(1)” for more information.

core.multiPackIndex

Use the multi-pack-index file to track multiple packfiles using a single index. See Section G.3.92, “git-multi-pack-index(1)” for more information. Defaults to true.

core.sparseCheckout

Enable "sparse checkout" feature. See Section G.3.135, “git-sparse-checkout(1)” for more information.

core.sparseCheckoutCone

Enables the "cone mode" of the sparse checkout feature. When the sparse-checkout file contains a limited set of patterns, this mode provides significant performance advantages. The "non-cone mode" can be requested to allow specifying more flexible patterns by setting this variable to *false*. See Section G.3.135, “git-sparse-checkout(1)” for more information.

core.abbrev

Set the length object names are abbreviated to. If unspecified or set to "auto", an appropriate value is computed based on the approximate number of packed objects in your repository, which hopefully is enough for abbreviated object names to stay unique for some time. If set to "no", no abbreviation is made and the object names are shown in their full length. The minimum length is 4.

core.maxTreeDepth

The maximum depth Git is willing to recurse while traversing a tree (e.g., "a/b/cde/f" has a depth of 4). This is a fail-safe to allow Git to abort cleanly, and should not generally need to be adjusted. The default is 4096.

credential.helper

Specify an external helper to be called when a username or password credential is needed; the helper may consult external storage to avoid prompting the user for the credentials. This is normally the name of a credential helper with possible arguments, but may also be an absolute path with arguments or, if preceded by *!*, shell commands.

Note that multiple helpers may be defined. See Section G.4.3, “gitcredentials(7)” for details and examples.

credential.useHttpPath

When acquiring credentials, consider the "path" component of an http or https URL to be important. Defaults to false. See Section G.4.3, “gitcredentials(7)” for more information.

credential.username

If no username is set for a network authentication, use this username by default. See credential.<context>.* below, and Section G.4.3, “gitcredentials(7)”.

credential.<url>.*

Any of the credential.* options above can be applied selectively to some credentials. For example, "credential.https://example.com.username" would set the default username only for https connections to example.com. See Section G.4.3, “gitcredentials(7)” for details on how URLs are matched.

credentialCache.ignoreSIGHUP

Tell git-credential-cache--daemon to ignore SIGHUP, instead of quitting.

credentialStore.lockTimeoutMS

The length of time, in milliseconds, for git-credential-store to retry when trying to lock the credentials file. A value of 0 means not to retry at all; -1 means to try indefinitely. Default is 1000 (i.e., retry for 1s).

diff.autoRefreshIndex

When using *git diff* to compare with work tree files, do not consider stat-only changes as changed. Instead, silently run *git update-index --refresh* to update the cached stat information for paths whose contents in the work tree match the contents in the index. This option defaults to true. Note that this affects only *git diff* Porcelain, and not lower level *diff* commands such as *git diff-files*.

diff.dirstat

A comma separated list of *--dirstat* parameters specifying the default behavior of the *--dirstat* option to Section G.3.44, “git-diff(1)” and friends. The defaults can be overridden on the command line (using *--dir-*

stat=<param1,param2,...>). The fallback defaults (when not changed by *diff.dirstat*) are *changes,noncumulative,3*. The following parameters are available:

changes

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

lines

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive *--dirstat* behavior than the *changes* behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other *--*stat* options.

files

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest *--dirstat* behavior, since it does not have to look at the file contents at all.

cumulative

Count changes in a child directory for the parent directory as well. Note that when using *cumulative*, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the *noncumulative* parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: *files,10,cumulative*.

diff.statNameWidth

Limit the width of the filename part in --stat output. If set, applies to all commands generating --stat output except format-patch.

diff.statGraphWidth

Limit the width of the graph part in --stat output. If set, applies to all commands generating --stat output except format-patch.

diff.context

Generate diffs with <n> lines of context instead of the default of 3. This value is overridden by the -U option.

diff.interHunkContext

Show the context between diff hunks, up to the specified number of lines, thereby fusing the hunks that are close to each other. This value serves as the default for the *--inter-hunk-context* command line option.

diff.external

If this config variable is set, diff generation is not performed using the internal diff machinery, but using the given command. Can be overridden with the GIT_EXTERNAL_DIFF environment variable. The command is called with parameters as described under "git Diffs" in Section G.3.1, “git(1)”. Note: if you want to use an

external diff program only on a subset of your files, you might want to use Section G.4.2, “gitattributes(5)” instead.

diff.ignoreSubmodules

Sets the default value of --ignore-submodules. Note that this affects only *git diff* Porcelain, and not lower level *diff* commands such as *git diff-files*. *git checkout* and *git switch* also honor this setting when reporting uncommitted changes. Setting it to *all* disables the submodule summary normally shown by *git commit* and *git status* when *status.submoduleSummary* is set unless it is overridden by using the --ignore-submodules command-line option. The *git submodule* commands are not affected by this setting. By default this is set to untracked so that any untracked submodules are ignored.

diff.mnemonicPrefix

If set, *git diff* uses a prefix pair that is different from the standard "a/" and "b/" depending on what is being compared. When this configuration is in effect, reverse diff output also swaps the order of the prefixes:

git diff

compares the (i)ndex and the (w)ork tree;

git diff HEAD

compares a (c)ommit and the (w)ork tree;

git diff --cached

compares a (c)ommit and the (i)ndex;

git diff HEAD:file1 file2

compares an (o)bject and a (w)ork tree entity;

git diff --no-index a b

compares two non-git things (1) and (2).

diff.noPrefix

If set, *git diff* does not show any source or destination prefix.

diff.srcPrefix

If set, *git diff* uses this source prefix. Defaults to "a/".

diff.dstPrefix

If set, *git diff* uses this destination prefix. Defaults to "b/".

diff.relative

If set to *true*, *git diff* does not show changes outside of the directory and show pathnames relative to the current directory.

diff.orderFile

File indicating how to order files within a diff. See the *-O* option to Section G.3.44, “git-diff(1)” for details. If *diff.orderFile* is a relative pathname, it is treated as relative to the top of the working tree.

diff.renameLimit

The number of files to consider in the exhaustive portion of copy/rename detection; equivalent to the *git diff* option *-l*. If not set, the default value is currently 1000. This setting has no effect if rename detection is turned off.

diff.renames

Whether and how Git detects renames. If set to "false", rename detection is disabled. If set to "true", basic rename detection is enabled. If set to "copies" or "copy", Git will detect copies, as well. Defaults to true. Note that this affects only *git diff* Porcelain like Section G.3.44, “git-diff(1)” and Section G.3.74, “git-log(1)”, and not lower level commands such as Section G.3.41, “git-diff-files(1)”.

diff.suppressBlankEmpty

A boolean to inhibit the standard behavior of printing a space before each empty output line. Defaults to false.

diff.submodule

Specify the format in which differences in submodules are shown. The "short" format just shows the names of the commits at the beginning and end of the range. The "log" format lists the commits in the range like Section G.3.141, “git-submodule(1)” *summary* does. The "diff" format shows an inline diff of the changed contents of the submodule. Defaults to "short".

diff.wordRegex

A POSIX Extended Regular Expression used to determine what is a "word" when performing word-by-word difference calculations. Character sequences that match the regular expression are "words", all other characters are **ignorable** whitespace.

diff.<driver>.command

The custom diff driver command. See Section G.4.2, “gitattributes(5)” for details.

diff.<driver>.xfuncname

The regular expression that the diff driver should use to recognize the hunk header. A built-in pattern may also be used. See Section G.4.2, “gitattributes(5)” for details.

diff.<driver>.binary

Set this option to true to make the diff driver treat files as binary. See Section G.4.2, “gitattributes(5)” for details.

diff.<driver>.textconv

The command that the diff driver should call to generate the text-converted version of a file. The result of the conversion is used to generate a human-readable diff. See Section G.4.2, “gitattributes(5)” for details.

diff.<driver>.wordRegex

The regular expression that the diff driver should use to split words in a line. See Section G.4.2, “gitattributes(5)” for details.

diff.<driver>.cachetextconv

Set this option to true to make the diff driver cache the text conversion outputs. See Section G.4.2, “gitattributes(5)” for details.

araxis

Use Araxis Merge (requires a graphical session)

bc

Use Beyond Compare (requires a graphical session)

bc3

Use Beyond Compare (requires a graphical session)

bc4

Use Beyond Compare (requires a graphical session)

codecompare

Use Code Compare (requires a graphical session)

deltawalker

Use DeltaWalker (requires a graphical session)

diffmerge

Use DiffMerge (requires a graphical session)

diffuse

Use Diffuse (requires a graphical session)

ecmerge

Use ECMerge (requires a graphical session)

emerge

Use Emacs' Emerge

examdiff

Use ExamDiff Pro (requires a graphical session)

guiffy

Use Guiffy's Diff Tool (requires a graphical session)

gvimdiff

Use gVim (requires a graphical session)

kdiff3

Use KDiff3 (requires a graphical session)

kompare

Use Kompare (requires a graphical session)

meld

Use Meld (requires a graphical session)

nvimdiff

Use Neovim

opendiff

Use FileMerge (requires a graphical session)

p4merge

Use HelixCore P4Merge (requires a graphical session)

smerge

Use Sublime Merge (requires a graphical session)

tkdiff

Use TkDiff (requires a graphical session)

vimdiff

Use Vim

winmerge

Use WinMerge (requires a graphical session)

xxdiff

Use xxdiff (requires a graphical session)

diff.indentHeuristic

Set this option to *false* to disable the default heuristics that shift diff hunk boundaries to make patches easier to read.

diff.algorithm

Choose a diff algorithm. The variants are as follows:

default, *myers*

The basic greedy diff algorithm. Currently, this is the default.

minimal

Spend extra time to make sure the smallest possible diff is produced.

patience

Use "patience diff" algorithm when generating patches.

histogram

This algorithm extends the patience algorithm to "support low-occurrence common elements".

diff.wsErrorHighlight

Highlight whitespace errors in the *context*, *old* or *new* lines of the diff. Multiple values are separated by comma, *none* resets previous values, *default* reset the list to *new* and *all* is a shorthand for *old,new,context*. The white-space errors are colored with *color.diff.whitespace*. The command line option *--ws-error-highlight=<kind>* overrides this setting.

diff.colorMoved

If set to either a valid *<mode>* or a true value, moved lines in a diff are colored differently, for details of valid modes see *--color-moved* in Section G.3.44, “git-diff(1)”. If simply set to true the default color mode will be used. When set to false, moved lines are not colored.

diff.colorMovedWS

When moved lines are colored using e.g. the *diff.colorMoved* setting, this option controls the *<mode>* how spaces are treated. For details of valid modes see *--color-moved-ws* in Section G.3.44, “git-diff(1)”.

diff.tool

Controls which diff tool is used by Section G.3.45, “git-difftool(1)”. This variable overrides the value configured in *merge.tool*. The list below shows the valid built-in values. Any other value is treated as a custom diff tool and requires that a corresponding difftool.<tool>.cmd variable is defined.

diff.guitool

Controls which diff tool is used by Section G.3.45, “git-difftool(1)” when the -g/--gui flag is specified. This variable overrides the value configured in *merge.guitool*. The list below shows the valid built-in values. Any other value is treated as a custom diff tool and requires that a corresponding difftool.<guitool>.cmd variable is defined.

difftool.<tool>.cmd

Specify the command to invoke the specified diff tool. The specified command is evaluated in shell with the following variables available: *LOCAL* is set to the name of the temporary file containing the contents of the diff pre-image and *REMOTE* is set to the name of the temporary file containing the contents of the diff post-image.

See the *--tool=<tool>* option in Section G.3.45, “git-difftool(1)” for more details.

difftool.<tool>.path

Override the path for the given tool. This is useful in case your tool is not in the PATH.

difftool.trustExitCode

Exit difftool if the invoked diff tool returns a non-zero exit status.

See the *--trust-exit-code* option in Section G.3.45, “git-difftool(1)” for more details.

difftool.prompt

Prompt before each invocation of the diff tool.

difftool.guiDefault

Set *true* to use the *diff.guitool* by default (equivalent to specifying the *--gui* argument), or *auto* to select *diff.guitool* or *diff.tool* depending on the presence of a *DISPLAY* environment variable value. The default is *false*, where the *--gui* argument must be provided explicitly for the *diff.guitool* to be used.

extensions.objectFormat

Specify the hash algorithm to use. The acceptable values are *sha1* and *sha256*. If not specified, *sha1* is assumed. It is an error to specify this key unless *core.repositoryFormatVersion* is 1.

Note that this setting should only be set by Section G.3.71, “git-init(1)” or Section G.3.24, “git-clone(1)”. Trying to change it after initialization will not work and will produce hard-to-diagnose issues.

extensions.compatObjectFormat

Specify a compatitbility hash algorithm to use. The acceptable values are *sha1* and *sha256*. The value specified must be different from the value of extensions.objectFormat. This allows client level interoperability between git repositories whose objectFormat matches this compatObjectFormat. In particular when fully implemented the pushes and pulls from a repository in whose objectFormat matches compatObjectFormat. As well as being able to use oids encoded in compatObjectFormat in addition to oids encoded with objectFormat to locally specify objects.

extensions.refStorage

Specify the ref storage format to use. The acceptable values are:

- *files* for loose files with packed-refs. This is the default.
- *reftable* for the reftable format. This format is experimental and its internals are subject to change.

  It is an error to specify this key unless *core.repositoryFormatVersion* is 1.

  Note that this setting should only be set by Section G.3.71, "git-init(1)" or Section G.3.24, "git-clone(1)". Trying to change it after initialization will not work and will produce hard-to-diagnose issues.

extensions.worktreeConfig

If enabled, then worktrees will load config settings from the *$GIT_DIR/config.worktree* file in addition to the *$GIT_COMMON_DIR/config* file. Note that *$GIT_COMMON_DIR* and *$GIT_DIR* are the same for the main working tree, while other working trees have *$GIT_DIR* equal to *$GIT_COMMON_DIR/worktrees/<id>/*. The settings in the *config.worktree* file will override settings from any other config files.

When enabling *extensions.worktreeConfig*, you must be careful to move certain values from the common config file to the main working tree's *config.worktree* file, if present:

- *core.worktree* must be moved from *$GIT_COMMON_DIR/config* to *$GIT_COMMON_DIR/config.worktree*.
- If *core.bare* is true, then it must be moved from *$GIT_COMMON_DIR/config* to *$GIT_COMMON_DIR/config.worktree*.

  It may also be beneficial to adjust the locations of *core.sparseCheckout* and *core.sparseCheckoutCone* depending on your desire for customizable sparse-checkout settings for each worktree. By default, the *git sparse-checkout* builtin enables *extensions.worktreeConfig*, assigns these config values on a per-worktree basis, and uses the *$GIT_DIR/info/sparse-checkout* file to specify the sparsity for each worktree independently. See Section G.3.135, "git-sparse-checkout(1)" for more details.

  For historical reasons, *extensions.worktreeConfig* is respected regardless of the *core.repositoryFormatVersion* setting.

fastimport.unpackLimit

If the number of objects imported by Section G.3.47, "git-fast-import(1)" is below this limit, then the objects will be unpacked into loose object files. However, if the number of imported objects equals or exceeds this limit, then the pack will be stored as a pack. Storing the pack from a fast-import can make the import operation complete faster, especially on slow filesystems. If not set, the value of *transfer.unpackLimit* is used instead.

feature.*

The config settings that start with *feature.* modify the defaults of a group of other config settings. These groups are created by the Git developer community as recommended defaults and are subject to change. In particular, new config options may be added with different defaults.

feature.experimental

Enable config options that are new to Git, and are being considered for future defaults. Config settings included here may be added or removed with each release, including minor version updates. These settings may have unintended interactions since they are so new. Please enable this setting if you are interested in providing feedback on experimental features. The new default values are:

- *fetch.negotiationAlgorithm=skipping* may improve fetch negotiation times by skipping more commits at a time, reducing the number of round trips.
- *pack.useBitmapBoundaryTraversal=true* may improve bitmap traversal times by walking fewer objects.
- *pack.allowPackReuse=multi* may improve the time it takes to create a pack by reusing objects from multiple packs instead of just one.

feature.manyFiles

Enable config options that optimize for repos with many files in the working directory. With many files, commands such as *git status* and *git checkout* may be slow and these new defaults improve performance:

- *index.skipHash=true* speeds up index writes by not computing a trailing checksum. Note that this will cause Git versions earlier than 2.13.0 to refuse to parse the index and Git versions earlier than 2.40.0 will report a corrupted index during *git fsck*.
- *index.version=4* enables path-prefix compression in the index.
- *core.untrackedCache=true* enables the untracked cache. This setting assumes that mtime is working on your machine.

fetch.recurseSubmodules

This option controls whether *git fetch* (and the underlying fetch in *git pull*) will recursively fetch into populated submodules. This option can be set either to a boolean value or to *on-demand*. Setting it to a boolean changes the behavior of fetch and pull to recurse unconditionally into submodules when set to true or to not recurse at all when set to false. When set to *on-demand*, fetch and pull will only recurse into a populated submodule when its superproject retrieves a commit that updates the submodule's reference. Defaults to *on-demand*, or to the value of *submodule.recurse* if set.

fetch.fsckObjects

If it is set to true, git-fetch-pack will check all fetched objects. See *transfer.fsckObjects* for what's checked. Defaults to false. If not set, the value of *transfer.fsckObjects* is used instead.

fetch.fsck.<msg-id>

Acts like *fsck.<msg-id>*, but is used by Section G.3.48, “git-fetch-pack(1)” instead of Section G.3.56, “git-fsck(1)”. See the *fsck.<msg-id>* documentation for details.

fetch.fsck.skipList

Acts like *fsck.skipList*, but is used by Section G.3.48, “git-fetch-pack(1)” instead of Section G.3.56, “git-fsck(1)”. See the *fsck.skipList* documentation for details.

fetch.unpackLimit

If the number of objects fetched over the Git native transfer is below this limit, then the objects will be unpacked into loose object files. However if the number of received objects equals or exceeds this limit then the received pack will be stored as a pack, after adding any missing delta bases. Storing the pack from a push can make the push operation complete faster, especially on slow filesystems. If not set, the value of *transfer.unpackLimit* is used instead.

fetch.prune

If true, fetch will automatically behave as if the *--prune* option was given on the command line. See also *remote.<name>.prune* and the PRUNING section of Section G.3.49, “git-fetch(1)”.

fetch.pruneTags

If true, fetch will automatically behave as if the *refs/tags/*:refs/tags/** refspec was provided when pruning, if not set already. This allows for setting both this option and *fetch.prune* to maintain a 1=1 mapping to upstream refs. See also *remote.<name>.pruneTags* and the PRUNING section of Section G.3.49, “git-fetch(1)”.

fetch.all

If true, fetch will attempt to update all available remotes. This behavior can be overridden by passing *--no-all* or by explicitly specifying one or more remote(s) to fetch from. Defaults to false.

fetch.output

Control how ref update status is printed. Valid values are *full* and *compact*. Default value is *full*. See the OUTPUT section in Section G.3.49, “git-fetch(1)” for details.

fetch.negotiationAlgorithm

Control how information about the commits in the local repository is sent when negotiating the contents of the packfile to be sent by the server. Set to "consecutive" to use an algorithm that walks over consecutive commits checking each one. Set to "skipping" to use an algorithm that skips commits in an effort to converge faster, but may result in a larger-than-necessary packfile; or set to "noop" to not send any information at all, which will almost certainly result in a larger-than-necessary packfile, but will skip the negotiation step. Set to "default" to override settings made previously and use the default behaviour. The default is normally "consecutive", but if *feature.experimental* is true, then the default is "skipping". Unknown values will cause *git fetch* to error out.

See also the *--negotiate-only* and *--negotiation-tip* options to Section G.3.49, “git-fetch(1)”.

fetch.showForcedUpdates

Set to false to enable *--no-show-forced-updates* in Section G.3.49, “git-fetch(1)” and Section G.3.102, “git-pull(1)” commands. Defaults to true.

fetch.parallel

Specifies the maximal number of fetch operations to be run in parallel at a time (submodules, or remotes when the *--multiple* option of Section G.3.49, “git-fetch(1)” is in effect).

A value of 0 will give some reasonable default. If unset, it defaults to 1.

For submodules, this setting can be overridden using the *submodule.fetchJobs* config setting.

fetch.writeCommitGraph

Set to true to write a commit-graph after every *git fetch* command that downloads a pack-file from a remote. Using the *--split* option, most executions will create a very small commit-graph file on top of the existing commit-graph file(s). Occasionally, these files will merge and the write may take longer. Having an updated commit-graph file helps performance of many Git commands, including *git merge-base*, *git push -f*, and *git log --graph*. Defaults to false.

fetch.bundleURI

This value stores a URI for downloading Git object data from a bundle URI before performing an incremental fetch from the origin Git server. This is similar to how the *--bundle-uri* option behaves in Section G.3.24, “git-clone(1)”. *git clone --bundle-uri* will set the *fetch.bundleURI* value if the supplied bundle URI contains a bundle list that is organized for incremental fetches.

If you modify this value and your repository has a *fetch.bundleCreationToken* value, then remove that *fetch.bundleCreationToken* value before fetching from the new bundle URI.

fetch.bundleCreationToken

When using *fetch.bundleURI* to fetch incrementally from a bundle list that uses the "creationToken" heuristic, this config value stores the maximum *creationToken* value of the downloaded bundles. This value is used to prevent downloading bundles in the future if the advertised *creationToken* is not strictly larger than this value.

The creation token values are chosen by the provider serving the specific bundle URI. If you modify the URI at *fetch.bundleURI*, then be sure to remove the value for the *fetch.bundleCreationToken* value before fetching.

filter.<driver>.clean

The command which is used to convert the content of a worktree file to a blob upon checkin. See Section G.4.2, “gitattributes(5)” for details.

filter.<driver>.smudge

The command which is used to convert the content of a blob object to a worktree file upon checkout. See Section G.4.2, “gitattributes(5)” for details.

format.attach

Enable multipart/mixed attachments as the default for *format-patch*. The value can also be a double quoted string which will enable attachments as the default and set the value as the boundary. See the --attach option in Section G.3.54, “git-format-patch(1)”. To countermand an earlier value, set it to an empty string.

format.from

Provides the default value for the *--from* option to format-patch. Accepts a boolean value, or a name and email address. If false, format-patch defaults to *--no-from*, using commit authors directly in the "From:" field of patch mails. If true, format-patch defaults to *--from*, using your committer identity in the "From:" field of patch mails and including a "From:" field in the body of the patch mail if different. If set to a non-boolean value, format-patch uses that value instead of your committer identity. Defaults to false.

format.forceInBodyFrom

Provides the default value for the *--[no-]force-in-body-from* option to format-patch. Defaults to false.

format.numbered

A boolean which can enable or disable sequence numbers in patch subjects. It defaults to "auto" which enables it only if there is more than one patch. It can be enabled or disabled for all messages by setting it to "true" or "false". See --numbered option in Section G.3.54, “git-format-patch(1)”.

format.headers

Additional email headers to include in a patch to be submitted by mail. See Section G.3.54, “git-format-patch(1)”.

format.to , format.cc

Additional recipients to include in a patch to be submitted by mail. See the --to and --cc options in Section G.3.54, “git-format-patch(1)”.

format.subjectPrefix

The default for format-patch is to output files with the *[PATCH]* subject prefix. Use this variable to change that prefix.

format.coverFromDescription

The default mode for format-patch to determine which parts of the cover letter will be populated using the branch's description. See the *--cover-from-description* option in Section G.3.54, “git-format-patch(1)”.

format.signature

The default for format-patch is to output a signature containing the Git version number. Use this variable to change that default. Set this variable to the empty string ("") to suppress signature generation.

format.signatureFile

Works just like format.signature except the contents of the file specified by this variable will be used as the signature.

format.suffix

The default for format-patch is to output files with the suffix *.patch*. Use this variable to change that suffix (make sure to include the dot if you want it).

format.encodeEmailHeaders

Encode email headers that have non-ASCII characters with "Q-encoding" (described in RFC 2047) for email transmission. Defaults to true.

format.pretty

The default pretty format for log/show/whatchanged command. See Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, Section G.3.158, “git-whatchanged(1)”.

format.thread

The default threading style for *git format-patch*. Can be a boolean value, or *shallow* or *deep*. *shallow* threading makes every mail a reply to the head of the series, where the head is chosen from the cover letter, the *--in-reply-to*, and the first patch mail, in this order. *deep* threading makes every mail a reply to the previous one. A true boolean value is the same as *shallow*, and a false value disables threading.

format.signOff

A boolean value which lets you enable the *-s/--signoff* option of format-patch by default. **Note:** Adding the *Signed-off-by* trailer to a patch should be a conscious act and means that you certify you have the rights to submit this work under the same open source license. Please see the *SubmittingPatches* document for further discussion.

format.coverLetter

A boolean that controls whether to generate a cover-letter when format-patch is invoked, but in addition can be set to "auto", to generate a cover-letter only when there's more than one patch. Default is false.

format.outputDirectory

Set a custom directory to store the resulting files instead of the current working directory. All directory components will be created.

format.filenameMaxLength

The maximum length of the output filenames generated by the *format-patch* command; defaults to 64. Can be overridden by the *--filename-max-length=<n>* command line option.

format.useAutoBase

A boolean value which lets you enable the *--base=auto* option of format-patch by default. Can also be set to "whenAble" to allow enabling *--base=auto* if a suitable base is available, but to skip adding base info otherwise without the format dying.

format.notes

Provides the default value for the *--notes* option to format-patch. Accepts a boolean value, or a ref which specifies where to get notes. If false, format-patch defaults to *--no-notes*. If true, format-patch defaults to *--notes*. If set to a non-boolean value, format-patch defaults to *--notes=<ref>*, where *ref* is the non-boolean value. Defaults to false.

If one wishes to use the ref *refs/notes/true*, please use that literal instead.

This configuration can be specified multiple times in order to allow multiple notes refs to be included. In that case, it will behave similarly to multiple *--[no-]notes[=]* options passed in. That is, a value of *true* will show the default notes, a value of *<ref>* will also show notes from that notes ref and a value of *false* will negate previous configurations and not show notes.

For example,

```
[format]
```

```
notes = true
notes = foo
notes = false
notes = bar
```

will only show notes from *refs/notes/bar*.

format.mboxrd

A boolean value which enables the robust "mboxrd" format when *--stdout* is in use to escape "^>+From " lines.

format.noprefix

If set, do not show any source or destination prefix in patches. This is equivalent to the *diff.noprefix* option used by *git diff* (but which is not respected by *format-patch*). Note that by setting this, the receiver of any patches you generate will have to apply them using the *-p0* option.

fsck.<msg-id>

During fsck git may find issues with legacy data which wouldn't be generated by current versions of git, and which wouldn't be sent over the wire if *transfer.fsckObjects* was set. This feature is intended to support working with legacy repositories containing such data.

Setting *fsck.<msg-id>* will be picked up by Section G.3.56, “git-fsck(1)”, but to accept pushes of such data set *receive.fsck.<msg-id>* instead, or to clone or fetch it set *fetch.fsck.<msg-id>*.

The rest of the documentation discusses *fsck.** for brevity, but the same applies for the corresponding *receive.fsck.** and *fetch.fsck.**. variables.

Unlike variables like *color.ui* and *core.editor*, the *receive.fsck.<msg-id>* and *fetch.fsck.<msg-id>* variables will not fall back on the *fsck.<msg-id>* configuration if they aren't set. To uniformly configure the same fsck settings in different circumstances, all three of them must be set to the same values.

When *fsck.<msg-id>* is set, errors can be switched to warnings and vice versa by configuring the *fsck.<msg-id>* setting where the *<msg-id>* is the fsck message ID and the value is one of *error*, *warn* or *ignore*. For convenience, fsck prefixes the error/warning with the message ID, e.g. "missingEmail: invalid author/committer line - missing email" means that setting *fsck.missingEmail = ignore* will hide that issue.

In general, it is better to enumerate existing objects with problems with *fsck.skipList*, instead of listing the kind of breakages these problematic objects share to be ignored, as doing the latter will allow new instances of the same breakages go unnoticed.

Setting an unknown *fsck.<msg-id>* value will cause fsck to die, but doing the same for *receive.fsck.<msg-id>* and *fetch.fsck.<msg-id>* will only cause git to warn.

See the *Fsck Messages* section of Section G.3.56, “git-fsck(1)” for supported values of *<msg-id>*.

fsck.skipList

The path to a list of object names (i.e. one unabbreviated SHA-1 per line) that are known to be broken in a non-fatal way and should be ignored. On versions of Git 2.20 and later, comments (#), empty lines, and any leading and trailing whitespace are ignored. Everything but a SHA-1 per line will error out on older versions.

This feature is useful when an established project should be accepted despite early commits containing errors that can be safely ignored, such as invalid committer email addresses. Note: corrupt objects cannot be skipped with this setting.

Like *fsck.<msg-id>* this variable has corresponding *receive.fsck.skipList* and *fetch.fsck.skipList* variants.

Unlike variables like *color.ui* and *core.editor* the *receive.fsck.skipList* and *fetch.fsck.skipList* variables will not fall back on the *fsck.skipList* configuration if they aren't set. To uniformly configure the same fsck settings in different circumstances, all three of them must be set to the same values.

Older versions of Git (before 2.20) documented that the object names list should be sorted. This was never a requirement; the object names could appear in any order, but when reading the list we tracked whether the list was sorted for the purposes of an internal binary search implementation, which could save itself some work with an already sorted list. Unless you had a humongous list there was no reason to go out of your way to pre-sort the list. After Git version 2.20 a hash implementation is used instead, so there's now no reason to pre-sort the list.

fsmonitor.allowRemote

By default, the fsmonitor daemon refuses to work with network-mounted repositories. Setting *fsmonitor.allowRemote* to *true* overrides this behavior. Only respected when *core.fsmonitor* is set to *true*.

fsmonitor.socketDir

This Mac OS-specific option, if set, specifies the directory in which to create the Unix domain socket used for communication between the fsmonitor daemon and various Git commands. The directory must reside on a native Mac OS filesystem. Only respected when *core.fsmonitor* is set to *true*.

gc.aggressiveDepth

The depth parameter used in the delta compression algorithm used by *git gc --aggressive*. This defaults to 50, which is the default for the *--depth* option when *--aggressive* isn't in use.

See the documentation for the *--depth* option in Section G.3.113, “git-repack(1)” for more details.

gc.aggressiveWindow

The window size parameter used in the delta compression algorithm used by *git gc --aggressive*. This defaults to 250, which is a much more aggressive window size than the default *--window* of 10.

See the documentation for the *--window* option in Section G.3.113, “git-repack(1)” for more details.

gc.auto

When there are approximately more than this many loose objects in the repository, *git gc --auto* will pack them. Some Porcelain commands use this command to perform a light-weight garbage collection from time to time. The default value is 6700.

Setting this to 0 disables not only automatic packing based on the number of loose objects, but also any other heuristic *git gc --auto* will otherwise use to determine if there's work to do, such as *gc.autoPackLimit*.

gc.autoPackLimit

When there are more than this many packs that are not marked with **.keep* file in the repository, *git gc --auto* consolidates them into one larger pack. The default value is 50. Setting this to 0 disables it. Setting *gc.auto* to 0 will also disable this.

See the *gc.bigPackThreshold* configuration variable below. When in use, it'll affect how the auto pack limit works.

gc.autoDetach

Make *git gc --auto* return immediately and run in the background if the system supports it. Default is true.

gc.bigPackThreshold

If non-zero, all non-cruft packs larger than this limit are kept when *git gc* is run. This is very similar to *--keep-largest-pack* except that all non-cruft packs that meet the threshold are kept, not just the largest pack. Defaults to zero. Common unit suffixes of *k*, *m*, or *g* are supported.

Note that if the number of kept packs is more than gc.autoPackLimit, this configuration variable is ignored, all packs except the base pack will be repacked. After this the number of packs should go below gc.autoPackLimit and gc.bigPackThreshold should be respected again.

If the amount of memory estimated for *git repack* to run smoothly is not available and *gc.bigPackThreshold* is not set, the largest pack will also be excluded (this is the equivalent of running *git gc* with *--keep-largest-pack*).

gc.writeCommitGraph

If true, then gc will rewrite the commit-graph file when Section G.3.58, “git-gc(1)” is run. When using *git gc --auto* the commit-graph will be updated if housekeeping is required. Default is true. See Section G.3.26, “git-commit-graph(1)” for details.

gc.logExpiry

If the file gc.log exists, then *git gc --auto* will print its content and exit with status zero instead of running unless that file is more than *gc.logExpiry* old. Default is "1.day". See *gc.pruneExpire* for more ways to specify its value.

gc.packRefs

Running *git pack-refs* in a repository renders it unclonable by Git versions prior to 1.5.1.2 over dumb transports such as HTTP. This variable determines whether *git gc* runs *git pack-refs*. This can be set to *notbare* to enable it within all non-bare repos or it can be set to a boolean value. The default is *true*.

gc.cruftPacks

Store unreachable objects in a cruft pack (see Section G.3.113, “git-repack(1)”) instead of as loose objects. The default is *true*.

gc.maxCruftSize

Limit the size of new cruft packs when repacking. When specified in addition to *--max-cruft-size*, the command line option takes priority. See the *--max-cruft-size* option of Section G.3.113, “git-repack(1)”.

gc.pruneExpire

When *git gc* is run, it will call *prune --expire 2.weeks.ago* (and *repack --cruft --cruft-expiration 2.weeks.ago* if using cruft packs via *gc.cruftPacks* or *--cruft*). Override the grace period with this config variable. The value "now" may be used to disable this grace period and always prune unreachable objects immediately, or "never" may be used to suppress pruning. This feature helps prevent corruption when *git gc* runs concurrently with another process writing to the repository; see the "NOTES" section of Section G.3.58, “git-gc(1)”.

gc.worktreePruneExpire

When *git gc* is run, it calls *git worktree prune --expire 3.months.ago*. This config variable can be used to set a different grace period. The value "now" may be used to disable the grace period and prune *$GIT_DIR/worktrees* immediately, or "never" may be used to suppress pruning.

gc.reflogExpire , gc.<pattern>.reflogExpire

git reflog expire removes reflog entries older than this time; defaults to 90 days. The value "now" expires all entries immediately, and "never" suppresses expiration altogether. With "<pattern>" (e.g. "refs/stash") in the middle the setting applies only to the refs that match the <pattern>.

gc.reflogExpireUnreachable , gc.<pattern>.reflogExpireUnreachable

git reflog expire removes reflog entries older than this time and are not reachable from the current tip; defaults to 30 days. The value "now" expires all entries immediately, and "never" suppresses expiration altogether. With "<pattern>" (e.g. "refs/stash") in the middle, the setting applies only to the refs that match the <pattern>.

These types of entries are generally created as a result of using *git commit --amend* or *git rebase* and are the commits prior to the amend or rebase occurring. Since these changes are not part of the current project most users will want to expire them sooner, which is why the default is more aggressive than *gc.reflogExpire*.

gc.recentObjectsHook

When considering whether or not to remove an object (either when generating a cruft pack or storing unreachable objects as loose), use the shell to execute the specified command(s). Interpret their output as object IDs which Git will consider as "recent", regardless of their age. By treating their mtimes as "now", any objects (and their descendants) mentioned in the output will be kept regardless of their true age.

Output must contain exactly one hex object ID per line, and nothing else. Objects which cannot be found in the repository are ignored. Multiple hooks are supported, but all must exit successfully, else the operation (either generating a cruft pack or unpacking unreachable objects) will be halted.

gc.repackFilter

When repacking, use the specified filter to move certain objects into a separate packfile. See the *--filter=<filter-spec>* option of Section G.3.113, “git-repack(1)”.

gc.repackFilterTo

When repacking and using a filter, see *gc.repackFilter*, the specified location will be used to create the packfile containing the filtered out objects. **WARNING:** The specified location should be accessible, using for example the Git alternates mechanism, otherwise the repo could be considered corrupt by Git as it migh not be able to access the objects in that packfile. See the *--filter-to=<dir>* option of Section G.3.113, “git-repack(1)” and the *objects/info/alternates* section of Section G.4.13, “gitrepository-layout(5)”.

gc.rerereResolved

Records of conflicted merge you resolved earlier are kept for this many days when *git rerere gc* is run. You can also use more human-readable "1.month.ago", etc. The default is 60 days. See Section G.3.117, “git-rerere(1)”.

gc.rerereUnresolved

Records of conflicted merge you have not resolved are kept for this many days when *git rerere gc* is run. You can also use more human-readable "1.month.ago", etc. The default is 15 days. See Section G.3.117, “git-rerere(1)”.

gitcvs.commitMsgAnnotation

Append this string to each commit message. Set to empty string to disable this feature. Defaults to "via git-CVS emulator".

gitcvs.enabled

Whether the CVS server interface is enabled for this repository. See Section G.3.37, “git-cvsserver(1)”.

gitcvs.logFile

Path to a log file where the CVS server interface well… logs various stuff. See Section G.3.37, “git-cvsserver(1)”.

gitcvs.usecrlfattr

If true, the server will look up the end-of-line conversion attributes for files to determine the *-k* modes to use. If the attributes force Git to treat a file as text, the *-k* mode will be left blank so CVS clients will treat it as text. If they suppress text conversion, the file will be set with *-kb* mode, which suppresses any newline munging the client might otherwise do. If the attributes do not allow the file type to be determined, then *gitcvs.allBinary* is used. See Section G.4.2, “gitattributes(5)”.

gitcvs.allBinary

This is used if *gitcvs.usecrlfattr* does not resolve the correct *-kb* mode to use. If true, all unresolved files are sent to the client in mode *-kb*. This causes the client to treat them as binary files, which suppresses any newline

munging it otherwise might do. Alternatively, if it is set to "guess", then the contents of the file are examined to decide if it is binary, similar to *core.autocrlf*.

gitcvs.dbName

Database used by git-cvsserver to cache revision information derived from the Git repository. The exact meaning depends on the used database driver, for SQLite (which is the default driver) this is a filename. Supports variable substitution (see Section G.3.37, “git-cvsserver(1)” for details). May not contain semicolons (*;*). Default: *%Ggitcvs.%m.sqlite*

gitcvs.dbDriver

Used Perl DBI driver. You can specify any available driver for this here, but it might not work. git-cvsserver is tested with *DBD::SQLite*, reported to work with *DBD::Pg*, and reported **not** to work with *DBD::mysql*. Experimental feature. May not contain double colons (*:*). Default: *SQLite*. See Section G.3.37, “git-cvsserver(1)”.

gitcvs.dbUser, gitcvs.dbPass

Database user and password. Only useful if setting *gitcvs.dbDriver*, since SQLite has no concept of database users and/or passwords. *gitcvs.dbUser* supports variable substitution (see Section G.3.37, “git-cvsserver(1)” for details).

gitcvs.dbTableNamePrefix

Database table name prefix. Prepended to the names of any database tables used, allowing a single database to be used for several repositories. Supports variable substitution (see Section G.3.37, “git-cvsserver(1)” for details). Any non-alphabetic characters will be replaced with underscores.

All gitcvs variables except for *gitcvs.usecrlfattr* and *gitcvs.allBinary* can also be specified as *gitcvs.<access_method>.<varname>* (where *access_method* is one of "ext" and "pserver") to make them apply only for the given access method.

gitweb.category , gitweb.description , gitweb.owner , gitweb.url

See Section G.4.16, “gitweb(1)” for description.

gitweb.avatar , gitweb.blame , gitweb.grep , gitweb.highlight , gitweb.patches , gitweb.pickaxe , gitweb.remote_heads , gitweb.showSizes , gitweb.snapshot

See Section G.4.17, “gitweb.conf(5)” for description.

gpg.program

Use this custom program instead of "*gpg*" found on *$PATH* when making or verifying a PGP signature. The program must support the same command-line interface as GPG, namely, to verify a detached signature, "*gpg --verify $signature - <$file*" is run, and the program is expected to signal a good signature by exiting with code 0. To generate an ASCII-armored detached signature, the standard input of "*gpg -bsau $key*" is fed with the contents to be signed, and the program is expected to send the result to its standard output.

gpg.format

Specifies which key format to use when signing with *--gpg-sign*. Default is "openpgp". Other possible values are "x509", "ssh".

See Section G.5.6, “gitformat-signature(5)” for the signature format, which differs based on the selected *gpg.format*.

gpg.<format>.program

Use this to customize the program used for the signing format you chose. (see *gpg.program* and *gpg.format*) *gpg.program* can still be used as a legacy synonym for *gpg.openpgp.program*. The default value for *gpg.x509.program* is "gpgsm" and *gpg.ssh.program* is "ssh-keygen".

gpg.minTrustLevel

Specifies a minimum trust level for signature verification. If this option is unset, then signature verification for merge operations requires a key with at least *marginal* trust. Other operations that perform signature verification require a key with at least *undefined* trust. Setting this option overrides the required trust-level for all operations. Supported values, in increasing order of significance:

- *undefined*
- *never*
- *marginal*
- *fully*
- *ultimate*

gpg.ssh.defaultKeyCommand

This command will be run when user.signingkey is not set and a ssh signature is requested. On successful exit a valid ssh public key prefixed with *key::* is expected in the first line of its output. This allows for a script doing a dynamic lookup of the correct public key when it is impractical to statically configure *user.signingKey*. For example when keys or SSH Certificates are rotated frequently or selection of the right key depends on external factors unknown to git.

gpg.ssh.allowedSignersFile

A file containing ssh public keys which you are willing to trust. The file consists of one or more lines of principals followed by an ssh public key. e.g.: *user1@example.com,user2@example.com ssh-rsa AAAAX1...* See ssh-keygen(1) "ALLOWED SIGNERS" for details. The principal is only used to identify the key and is available when verifying a signature.

SSH has no concept of trust levels like gpg does. To be able to differentiate between valid signatures and trusted signatures the trust level of a signature verification is set to *fully* when the public key is present in the allowedSignersFile. Otherwise the trust level is *undefined* and git verify-commit/tag will fail.

This file can be set to a location outside of the repository and every developer maintains their own trust store. A central repository server could generate this file automatically from ssh keys with push access to verify the code against. In a corporate setting this file is probably generated at a global location from automation that already handles developer ssh keys.

A repository that only allows signed commits can store the file in the repository itself using a path relative to the top-level of the working tree. This way only committers with an already valid key can add or change keys in the keyring.

Since OpensSSH 8.8 this file allows specifying a key lifetime using valid-after & valid-before options. Git will mark signatures as valid if the signing key was valid at the time of the signature's creation. This allows users to change a signing key without invalidating all previously made signatures.

Using a SSH CA key with the cert-authority option (see ssh-keygen(1) "CERTIFICATES") is also valid.

gpg.ssh.revocationFile

Either a SSH KRL or a list of revoked public keys (without the principal prefix). See ssh-keygen(1) for details. If a public key is found in this file then it will always be treated as having trust level "never" and signatures will show as invalid.

grep.lineNumber

If set to true, enable *-n* option by default.

grep.column

If set to true, enable the *--column* option by default.

grep.patternType

Set the default matching behavior. Using a value of *basic*, *extended*, *fixed*, or *perl* will enable the *--basic-regexp*, *--extended-regexp*, *--fixed-strings*, or *--perl-regexp* option accordingly, while the value *default* will use the *grep.extendedRegexp* option to choose between *basic* and *extended*.

grep.extendedRegexp

If set to true, enable *--extended-regexp* option by default. This option is ignored when the *grep.patternType* option is set to a value other than *default*.

grep.threads

Number of grep worker threads to use. If unset (or set to 0), Git will use as many threads as the number of logical cores available.

grep.fullName

If set to true, enable *--full-name* option by default.

grep.fallbackToNoIndex

If set to true, fall back to *git grep --no-index* if *git grep* is executed outside of a git repository. Defaults to false.

gui.commitMsgWidth

Defines how wide the commit message window is in the Section G.3.61, “git-gui(1)”. "75" is the default.

gui.diffContext

Specifies how many context lines should be used in calls to diff made by the Section G.3.61, “git-gui(1)”. The default is "5".

gui.displayUntracked

Determines if Section G.3.61, “git-gui(1)” shows untracked files in the file list. The default is "true".

gui.encoding

Specifies the default character encoding to use for displaying of file contents in Section G.3.61, “git-gui(1)” and Section G.4.8, “gitk(1)”. It can be overridden by setting the *encoding* attribute for relevant files (see Section G.4.2, “gitattributes(5)”). If this option is not set, the tools default to the locale encoding.

gui.matchTrackingBranch

Determines if new branches created with Section G.3.61, “git-gui(1)” should default to tracking remote branches with matching names or not. Default: "false".

gui.newBranchTemplate

Is used as a suggested name when creating new branches using the Section G.3.61, “git-gui(1)”.

gui.pruneDuringFetch

"true" if Section G.3.61, “git-gui(1)” should prune remote-tracking branches when performing a fetch. The default value is "false".

gui.trustmtime

Determines if Section G.3.61, “git-gui(1)” should trust the file modification timestamp or not. By default the timestamps are not trusted.

gui.spellingDictionary

Specifies the dictionary used for spell checking commit messages in the Section G.3.61, “git-gui(1)”. When set to "none" spell checking is turned off.

gui.fastCopyBlame

If true, *git gui blame* uses *-C* instead of *-C -C* for original location detection. It makes blame significantly faster on huge repositories at the expense of less thorough copy detection.

gui.copyBlameThreshold

Specifies the threshold to use in *git gui blame* original location detection, measured in alphanumeric characters. See the Section G.3.9, “git-blame(1)” manual for more information on copy detection.

gui.blamehistoryctx

Specifies the radius of history context in days to show in Section G.4.8, “gitk(1)” for the selected commit, when the *Show History Context* menu item is invoked from *git gui blame*. If this variable is set to zero, the whole history is shown.

guitool.<name>.cmd

Specifies the shell command line to execute when the corresponding item of the Section G.3.61, “git-gui(1)” *Tools* menu is invoked. This option is mandatory for every tool. The command is executed from the root of the working directory, and in the environment it receives the name of the tool as *GIT_GUITOOL*, the name of the currently selected file as *FILENAME*, and the name of the current branch as *CUR_BRANCH* (if the head is detached, *CUR_BRANCH* is empty).

guitool.<name>.needsFile

Run the tool only if a diff is selected in the GUI. It guarantees that *FILENAME* is not empty.

guitool.<name>.noConsole

Run the command silently, without creating a window to display its output.

guitool.<name>.noRescan

Don't rescan the working directory for changes after the tool finishes execution.

guitool.<name>.confirm

Show a confirmation dialog before actually running the tool.

guitool.<name>.argPrompt

Request a string argument from the user, and pass it to the tool through the *ARGS* environment variable. Since requesting an argument implies confirmation, the *confirm* option has no effect if this is enabled. If the option is set to *true*, *yes*, or *1*, the dialog uses a built-in generic prompt; otherwise the exact value of the variable is used.

guitool.<name>.revPrompt

Request a single valid revision from the user, and set the *REVISION* environment variable. In other aspects this option is similar to *argPrompt*, and can be used together with it.

guitool.<name>.revUnmerged

Show only unmerged branches in the *revPrompt* subdialog. This is useful for tools similar to merge or rebase, but not for things like checkout or reset.

guitool.<name>.title

Specifies the title to use for the prompt dialog. The default is the tool name.

guitool.<name>.prompt

Specifies the general prompt string to display at the top of the dialog, before subsections for *argPrompt* and *revPrompt*. The default value includes the actual command.

help.browser

Specify the browser that will be used to display help in the *web* format. See Section G.3.63, “git-help(1)”.

help.format

Override the default help format used by Section G.3.63, “git-help(1)”. Values *man*, *info*, *web* and *html* are supported. *man* is the default. *web* and *html* are the same.

help.autoCorrect

If git detects typos and can identify exactly one valid command similar to the error, git will try to suggest the correct command or even run the suggestion automatically. Possible config values are:

- 0 (default): show the suggested command.
- positive number: run the suggested command after specified deciseconds (0.1 sec).
- "immediate": run the suggested command immediately.
- "prompt": show the suggestion and prompt for confirmation to run the command.
- "never": don't run or show any suggested command.

help.htmlPath

Specify the path where the HTML documentation resides. File system paths and URLs are supported. HTML pages will be prefixed with this path when help is displayed in the *web* format. This defaults to the documentation path of your Git installation.

http.proxy

Override the HTTP proxy, normally configured using the *http_proxy*, *https_proxy*, and *all_proxy* environment variables (see *curl(1)*). In addition to the syntax understood by curl, it is possible to specify a proxy string with a user name but no password, in which case git will attempt to acquire one in the same way it does for other credentials. See Section G.4.3, “gitcredentials(7)” for more information. The syntax thus is *[protocol://][user[:password]@]proxyhost[:port]*. This can be overridden on a per-remote basis; see remote.<name>.proxy

http.proxyAuthMethod

Set the method with which to authenticate against the HTTP proxy. This only takes effect if the configured proxy string contains a user name part (i.e. is of the form *user@host* or *user@host:port*). This can be overridden on a per-remote basis; see *remote.<name>.proxyAuthMethod*. Both can be overridden by the *GIT_HTTP_PROXY_AUTHMETHOD* environment variable. Possible values are:

- *anyauth* - Automatically pick a suitable authentication method. It is assumed that the proxy answers an unauthenticated request with a 407 status code and one or more Proxy-authenticate headers with supported authentication methods. This is the default.
- *basic* - HTTP Basic authentication
- *digest* - HTTP Digest authentication; this prevents the password from being transmitted to the proxy in clear text
- *negotiate* - GSS-Negotiate authentication (compare the --negotiate option of *curl(1)*)
- *ntlm* - NTLM authentication (compare the --ntlm option of *curl(1)*)

http.proxySSLCert

The pathname of a file that stores a client certificate to use to authenticate with an HTTPS proxy. Can be overridden by the *GIT_PROXY_SSL_CERT* environment variable.

http.proxySSLKey

The pathname of a file that stores a private key to use to authenticate with an HTTPS proxy. Can be overridden by the *GIT_PROXY_SSL_KEY* environment variable.

http.proxySSLCertPasswordProtected

Enable Git's password prompt for the proxy SSL certificate. Otherwise OpenSSL will prompt the user, possibly many times, if the certificate or private key is encrypted. Can be overridden by the *GIT_PROXY_SSL_CERT_PASSWORD_PROTECTED* environment variable.

http.proxySSLCAInfo

Pathname to the file containing the certificate bundle that should be used to verify the proxy with when using an HTTPS proxy. Can be overridden by the *GIT_PROXY_SSL_CAINFO* environment variable.

http.emptyAuth

Attempt authentication without seeking a username or password. This can be used to attempt GSS-Negotiate authentication without specifying a username in the URL, as libcurl normally requires a username for authentication.

http.delegation

Control GSSAPI credential delegation. The delegation is disabled by default in libcurl since version 7.21.7. Set parameter to tell the server what it is allowed to delegate when it comes to user credentials. Used with GSS/kerberos. Possible values are:

- *none* - Don't allow any delegation.
- *policy* - Delegates if and only if the OK-AS-DELEGATE flag is set in the Kerberos service ticket, which is a matter of realm policy.
- *always* - Unconditionally allow the server to delegate.

http.extraHeader

Pass an additional HTTP header when communicating with a server. If more than one such entry exists, all of them are added as extra headers. To allow overriding the settings inherited from the system config, an empty value will reset the extra headers to the empty list.

http.cookieFile

The pathname of a file containing previously stored cookie lines, which should be used in the Git http session, if they match the server. The file format of the file to read cookies from should be plain HTTP headers or the Netscape/Mozilla cookie file format (see *curl(1)*). NOTE that the file specified with http.cookieFile is used only as input unless http.saveCookies is set.

http.saveCookies

If set, store cookies received during requests to the file specified by http.cookieFile. Has no effect if http.cookieFile is unset.

http.version

Use the specified HTTP protocol version when communicating with a server. If you want to force the default. The available and default version depend on libcurl. Currently the possible values of this option are:

- HTTP/2
- HTTP/1.1

http.curloptResolve

Hostname resolution information that will be used first by libcurl when sending HTTP requests. This information should be in one of the following formats:

- [+]HOST:PORT:ADDRESS[,ADDRESS]
- -HOST:PORT

The first format redirects all requests to the given *HOST:PORT* to the provided *ADDRESS*(s). The second format clears all previous config values for that *HOST:PORT* combination. To allow easy overriding of all the settings inherited from the system config, an empty value will reset all resolution information to the empty list.

http.sslVersion

The SSL version to use when negotiating an SSL connection, if you want to force the default. The available and default version depend on whether libcurl was built against NSS or OpenSSL and the particular configuration of the crypto library in use. Internally this sets the *CURLOPT_SSL_VERSION* option; see the libcurl documentation for more details on the format of this option and for the ssl version supported. Currently the possible values of this option are:

- sslv2
- sslv3
- tlsv1
- tlsv1.0
- tlsv1.1
- tlsv1.2
- tlsv1.3

Can be overridden by the *GIT_SSL_VERSION* environment variable. To force git to use libcurl's default ssl version and ignore any explicit http.sslversion option, set *GIT_SSL_VERSION* to the empty string.

http.sslCipherList

A list of SSL ciphers to use when negotiating an SSL connection. The available ciphers depend on whether libcurl was built against NSS or OpenSSL and the particular configuration of the crypto library in use. Internally this sets the *CURLOPT_SSL_CIPHER_LIST* option; see the libcurl documentation for more details on the format of this list.

Can be overridden by the *GIT_SSL_CIPHER_LIST* environment variable. To force git to use libcurl's default cipher list and ignore any explicit http.sslCipherList option, set *GIT_SSL_CIPHER_LIST* to the empty string.

http.sslVerify

Whether to verify the SSL certificate when fetching or pushing over HTTPS. Defaults to true. Can be overridden by the *GIT_SSL_NO_VERIFY* environment variable.

http.sslCert

File containing the SSL certificate when fetching or pushing over HTTPS. Can be overridden by the *GIT_SSL_CERT* environment variable.

http.sslKey

File containing the SSL private key when fetching or pushing over HTTPS. Can be overridden by the *GIT_SSL_KEY* environment variable.

http.sslCertPasswordProtected

Enable Git's password prompt for the SSL certificate. Otherwise OpenSSL will prompt the user, possibly many times, if the certificate or private key is encrypted. Can be overridden by the *GIT_SSL_CERT_PASSWORD_PROTECTED* environment variable.

http.sslCAInfo

File containing the certificates to verify the peer with when fetching or pushing over HTTPS. Can be overridden by the *GIT_SSL_CAINFO* environment variable.

http.sslCAPath

Path containing files with the CA certificates to verify the peer with when fetching or pushing over HTTPS. Can be overridden by the *GIT_SSL_CAPATH* environment variable.

http.sslBackend

Name of the SSL backend to use (e.g. "openssl" or "schannel"). This option is ignored if cURL lacks support for choosing the SSL backend at runtime.

http.schannelCheckRevoke

Used to enforce or disable certificate revocation checks in cURL when http.sslBackend is set to "schannel". Defaults to *true* if unset. Only necessary to disable this if Git consistently errors and the message is about checking the revocation status of a certificate. This option is ignored if cURL lacks support for setting the relevant SSL option at runtime.

http.schannelUseSSLCAInfo

As of cURL v7.60.0, the Secure Channel backend can use the certificate bundle provided via *http.sslCAInfo*, but that would override the Windows Certificate Store. Since this is not desirable by default, Git will tell cURL not to use that bundle by default when the *schannel* backend was configured via *http.sslBackend*, unless *http.schannelUseSSLCAInfo* overrides this behavior.

http.pinnedPubkey

Public key of the https service. It may either be the filename of a PEM or DER encoded public key file or a string starting with *sha256//* followed by the base64 encoded sha256 hash of the public key. See also libcurl *CURLOPT_PINNEDPUBLICKEY*. git will exit with an error if this option is set but not supported by cURL.

http.sslTry

Attempt to use AUTH SSL/TLS and encrypted data transfers when connecting via regular FTP protocol. This might be needed if the FTP server requires it for security reasons or you wish to connect securely whenever remote FTP server supports it. Default is false since it might trigger certificate verification errors on misconfigured servers.

http.maxRequests

How many HTTP requests to launch in parallel. Can be overridden by the *GIT_HTTP_MAX_REQUESTS* environment variable. Default is 5.

http.minSessions

The number of curl sessions (counted across slots) to be kept across requests. They will not be ended with curl_easy_cleanup() until http_cleanup() is invoked. If USE_CURL_MULTI is not defined, this value will be capped at 1. Defaults to 1.

http.postBuffer

Maximum size in bytes of the buffer used by smart HTTP transports when POSTing data to the remote system. For requests larger than this buffer size, HTTP/1.1 and Transfer-Encoding: chunked is used to avoid creating a massive pack file locally. Default is 1 MiB, which is sufficient for most requests.

Note that raising this limit is only effective for disabling chunked transfer encoding and therefore should be used only where the remote server or a proxy only supports HTTP/1.0 or is noncompliant with the HTTP standard. Raising this is not, in general, an effective solution for most push problems, but can increase memory consumption significantly since the entire buffer is allocated even for small pushes.

http.lowSpeedLimit, http.lowSpeedTime

If the HTTP transfer speed, in bytes per second, is less than *http.lowSpeedLimit* for longer than *http.lowSpeedTime* seconds, the transfer is aborted. Can be overridden by the *GIT_HTTP_LOW_SPEED_LIMIT* and *GIT_HTTP_LOW_SPEED_TIME* environment variables.

http.noEPSV

A boolean which disables using of EPSV ftp command by curl. This can be helpful with some "poor" ftp servers which don't support EPSV mode. Can be overridden by the *GIT_CURL_FTP_NO_EPSV* environment variable. Default is false (curl will use EPSV).

http.userAgent

The HTTP USER_AGENT string presented to an HTTP server. The default value represents the version of the Git client such as git/1.7.1. This option allows you to override this value to a more common value such as Mozilla/4.0. This may be necessary, for instance, if connecting through a firewall that restricts HTTP connections to a set of common USER_AGENT strings (but not including those like git/1.7.1). Can be overridden by the *GIT_HTTP_USER_AGENT* environment variable.

http.followRedirects

Whether git should follow HTTP redirects. If set to *true*, git will transparently follow any redirect issued by a server it encounters. If set to *false*, git will treat all redirects as errors. If set to *initial*, git will follow redirects only for the initial request to a remote, but not for subsequent follow-up HTTP requests. Since git uses the redirected URL as the base for the follow-up requests, this is generally sufficient. The default is *initial*.

http.<url>.*

Any of the http.* options above can be applied selectively to some URLs. For a config key to match a URL, each element of the config key is compared to that of the URL, in the following order:

1. Scheme (e.g., *https* in *https://example.com/*). This field must match exactly between the config key and the URL.

2. Host/domain name (e.g., *example.com* in *https://example.com/*). This field must match between the config key and the URL. It is possible to specify a * as part of the host name to match all subdomains at this level. *https://*.example.com/* for example would match *https://foo.example.com/*, but not *https://foo.bar.example.com/*.

3. Port number (e.g., *8080* in *http://example.com:8080/*). This field must match exactly between the config key and the URL. Omitted port numbers are automatically converted to the correct default for the scheme before matching.

4. Path (e.g., *repo.git* in *https://example.com/repo.git*). The path field of the config key must match the path field of the URL either exactly or as a prefix of slash-delimited path elements. This means a config key with path *foo/* matches URL path *foo/bar*. A prefix can only match on a slash (/) boundary. Longer matches take precedence (so a config key with path *foo/bar* is a better match to URL path *foo/bar* than a config key with just path *foo/*).

5. User name (e.g., *user* in *https://user@example.com/repo.git*). If the config key has a user name it must match the user name in the URL exactly. If the config key does not have a user name, that config key will match a URL with any user name (including none), but at a lower precedence than a config key with a user name.

The list above is ordered by decreasing precedence; a URL that matches a config key's path is preferred to one that matches its user name. For example, if the URL is *https://user@example.com/foo/bar* a config key match of *https://example.com/foo* will be preferred over a config key match of *https://user@example.com*.

All URLs are normalized before attempting any matching (the password part, if embedded in the URL, is always ignored for matching purposes) so that equivalent URLs that are simply spelled differently will match properly. Environment variable settings always override any matches. The URLs that are matched against are those given directly to Git commands. This means any URLs visited as a result of a redirection do not participate in matching.

i18n.commitEncoding

Character encoding the commit messages are stored in; Git itself does not care per se, but this information is necessary e.g. when importing commits from emails or in the gitk graphical history browser (and possibly in other places in the future or in other porcelains). See e.g. Section G.3.78, “git-mailinfo(1)”. Defaults to *utf-8*.

i18n.logOutputEncoding

Character encoding the commit messages are converted to when running *git log* and friends.

imap.folder

The folder to drop the mails into, which is typically the Drafts folder. For example: "INBOX.Drafts", "INBOX/Drafts" or "[Gmail]/Drafts". Required.

imap.tunnel

Command used to set up a tunnel to the IMAP server through which commands will be piped instead of using a direct network connection to the server. Required when imap.host is not set.

imap.host

A URL identifying the server. Use an *imap://* prefix for non-secure connections and an *imaps://* prefix for secure connections. Ignored when imap.tunnel is set, but required otherwise.

imap.user

The username to use when logging in to the server.

imap.pass

The password to use when logging in to the server.

imap.port

An integer port number to connect to on the server. Defaults to 143 for imap:// hosts and 993 for imaps:// hosts. Ignored when imap.tunnel is set.

imap.sslverify

A boolean to enable/disable verification of the server certificate used by the SSL/TLS connection. Default is *true*. Ignored when imap.tunnel is set.

imap.preformattedHTML

A boolean to enable/disable the use of html encoding when sending a patch. An html encoded patch will be bracketed with <pre> and have a content type of text/html. Ironically, enabling this option causes Thunderbird to send the patch as a plain/text, format=fixed email. Default is *false*.

imap.authMethod

Specify the authentication method for authenticating with the IMAP server. If Git was built with the NO_CURL option, or if your curl version is older than 7.34.0, or if you're running git-imap-send with the *--no-curl* option, the only supported method is *CRAM-MD5*. If this is not set then *git imap-send* uses the basic IMAP plaintext LOGIN command.

include.path , includeIf.<condition>.path

Special variables to include other configuration files. See the "CONFIGURATION FILE" section in the main Section G.3.29, “git-config(1)” documentation, specifically the "Includes" and "Conditional Includes" subsections.

index.recordEndOfIndexEntries

Specifies whether the index file should include an "End Of Index Entry" section. This reduces index load time on multiprocessor machines but produces a message "ignoring EOIE extension" when reading the index using Git versions before 2.20. Defaults to *true* if index.threads has been explicitly enabled, *false* otherwise.

index.recordOffsetTable

Specifies whether the index file should include an "Index Entry Offset Table" section. This reduces index load time on multiprocessor machines but produces a message "ignoring IEOT extension" when reading the index using Git versions before 2.20. Defaults to *true* if index.threads has been explicitly enabled, *false* otherwise.

index.sparse

When enabled, write the index using sparse-directory entries. This has no effect unless *core.sparseCheckout* and *core.sparseCheckoutCone* are both enabled. Defaults to *false*.

index.threads

Specifies the number of threads to spawn when loading the index. This is meant to reduce index load time on multiprocessor machines. Specifying 0 or *true* will cause Git to auto-detect the number of CPUs and set the number of threads accordingly. Specifying 1 or *false* will disable multithreading. Defaults to *true*.

index.version

Specify the version with which new index files should be initialized. This does not affect existing repositories. If *feature.manyFiles* is enabled, then the default is 4.

index.skipHash

When enabled, do not compute the trailing hash for the index file. This accelerates Git commands that manipulate the index, such as *git add*, *git commit*, or *git status*. Instead of storing the checksum, write a trailing set of bytes with value zero, indicating that the computation was skipped.

If you enable *index.skipHash*, then Git clients older than 2.13.0 will refuse to parse the index and Git clients older than 2.40.0 will report an error during *git fsck*.

init.templateDir

Specify the directory from which templates will be copied. (See the "TEMPLATE DIRECTORY" section of Section G.3.71, “git-init(1)”.)

init.defaultBranch

Allows overriding the default branch name e.g. when initializing a new repository.

instaweb.browser

Specify the program that will be used to browse your working repository in gitweb. See Section G.3.72, “git-instaweb(1)”.

instaweb.httpd

The HTTP daemon command-line to start gitweb on your working repository. See Section G.3.72, “git-instaweb(1)”.

instaweb.local

If true the web server started by Section G.3.72, “git-instaweb(1)” will be bound to the local IP (127.0.0.1).

instaweb.modulePath

The default module path for Section G.3.72, “git-instaweb(1)” to use instead of /usr/lib/apache2/modules. Only used if httpd is Apache.

instaweb.port

The port number to bind the gitweb httpd to. See Section G.3.72, “git-instaweb(1)”.

interactive.singleKey

When set to true, allow the user to provide one-letter input with a single key (i.e., without hitting the Enter key) in interactive commands. This is currently used by the *--patch* mode of Section G.3.2, “git-add(1)”, Section G.3.19, “git-checkout(1)”, Section G.3.119, “git-restore(1)”, Section G.3.28, “git-commit(1)”, Section G.3.118, “git-reset(1)”, and Section G.3.137, “git-stash(1)”.

interactive.diffFilter

When an interactive command (such as *git add --patch*) shows a colorized diff, git will pipe the diff through the shell command defined by this configuration variable. The command may mark up the diff further for human consumption, provided that it retains a one-to-one correspondence with the lines in the original diff. Defaults to disabled (no filtering).

log.abbrevCommit

If true, makes Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, and Section G.3.158, “git-whatchanged(1)” assume *--abbrev-commit*. You may override this option with *--no-abbrev-commit*.

log.date

Set the default date-time mode for the *log* command. Setting a value for log.date is similar to using *git log*'s *--date* option. See Section G.3.74, “git-log(1)” for details.

If the format is set to "auto:foo" and the pager is in use, format "foo" will be used for the date format. Otherwise, "default" will be used.

log.decorate

Print out the ref names of any commits that are shown by the log command. If *short* is specified, the ref name prefixes *refs/heads/*, *refs/tags/* and *refs/remotes/* will not be printed. If *full* is specified, the full ref name (including prefix) will be printed. If *auto* is specified, then if the output is going to a terminal, the ref names are shown as if *short* were given, otherwise no ref names are shown. This is the same as the *--decorate* option of the *git log*.

log.initialDecorationSet

By default, *git log* only shows decorations for certain known ref namespaces. If *all* is specified, then show all refs as decorations.

log.excludeDecoration

Exclude the specified patterns from the log decorations. This is similar to the *--decorate-refs-exclude* command-line option, but the config option can be overridden by the *--decorate-refs* option.

log.diffMerges

Set diff format to be used when *--diff-merges=on* is specified, see *--diff-merges* in Section G.3.74, “git-log(1)” for details. Defaults to *separate*.

log.follow

If *true*, *git log* will act as if the *--follow* option was used when a single <path> is given. This has the same limitations as *--follow*, i.e. it cannot be used to follow multiple files and does not work well on non-linear history.

log.graphColors

A list of colors, separated by commas, that can be used to draw history lines in *git log --graph*.

log.showRoot

If true, the initial commit will be shown as a big creation event. This is equivalent to a diff against an empty tree. Tools like Section G.3.74, “git-log(1)” or Section G.3.158, “git-whatchanged(1)”, which normally hide the root commit will now show it. True by default.

log.showSignature

If true, makes Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, and Section G.3.158, “git-whatchanged(1)” assume *--show-signature*.

log.mailmap

If true, makes Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, and Section G.3.158, “git-whatchanged(1)” assume *--use-mailmap*, otherwise assume *--no-use-mailmap*. True by default.

lsrefs.unborn

May be "advertise" (the default), "allow", or "ignore". If "advertise", the server will respond to the client sending "unborn" (as described in Section G.5.11, “gitprotocol-v2(5)”) and will advertise support for this feature during the protocol v2 capability advertisement. "allow" is the same as "advertise" except that the server will not advertise support for this feature; this is useful for load-balanced servers that cannot be updated atomically (for example), since the administrator could configure "allow", then after a delay, configure "advertise".

mailinfo.scissors

If true, makes Section G.3.78, “git-mailinfo(1)” (and therefore Section G.3.3, “git-am(1)”) act by default as if the --scissors option was provided on the command-line. When active, this feature removes everything from the message body before a scissors line (i.e. consisting mainly of ">8", "8<" and "-").

mailmap.file

The location of an augmenting mailmap file. The default mailmap, located in the root of the repository, is loaded first, then the mailmap file pointed to by this variable. The location of the mailmap file may be in a repository subdirectory, or somewhere outside of the repository itself. See Section G.3.130, “git-shortlog(1)” and Section G.3.9, “git-blame(1)”.

mailmap.blob

Like *mailmap.file*, but consider the value as a reference to a blob in the repository. If both *mailmap.file* and *mailmap.blob* are given, both are parsed, with entries from *mailmap.file* taking precedence. In a bare repository, this defaults to *HEAD:.mailmap*. In a non-bare repository, it defaults to empty.

maintenance.auto

This boolean config option controls whether some commands run *git maintenance run --auto* after doing their normal work. Defaults to true.

maintenance.strategy

This string config option provides a way to specify one of a few recommended schedules for background maintenance. This only affects which tasks are run during *git maintenance run --schedule=X* commands, provided no *--task=<task>* arguments are provided. Further, if a *maintenance.<task>.schedule* config value is set, then that value is used instead of the one provided by *maintenance.strategy*. The possible strategy strings are:

- *none*: This default setting implies no tasks are run at any schedule.
- *incremental*: This setting optimizes for performing small maintenance activities that do not delete any data. This does not schedule the *gc* task, but runs the *prefetch* and *commit-graph* tasks hourly, the *loose-objects* and *incremental-repack* tasks daily, and the *pack-refs* task weekly.

maintenance.<task>.enabled

This boolean config option controls whether the maintenance task with name *<task>* is run when no *--task* option is specified to *git maintenance run*. These config values are ignored if a *--task* option exists. By default, only *maintenance.gc.enabled* is true.

maintenance.<task>.schedule

This config option controls whether or not the given *<task>* runs during a *git maintenance run --schedule=<frequency>* command. The value must be one of "hourly", "daily", or "weekly".

maintenance.commit-graph.auto

This integer config option controls how often the *commit-graph* task should be run as part of *git maintenance run --auto*. If zero, then the *commit-graph* task will not run with the *--auto* option. A negative value will force the task to run every time. Otherwise, a positive value implies the command should run when the number of reachable commits that are not in the commit-graph file is at least the value of *maintenance.commit-graph.auto*. The default value is 100.

maintenance.loose-objects.auto

This integer config option controls how often the *loose-objects* task should be run as part of *git maintenance run --auto*. If zero, then the *loose-objects* task will not run with the *--auto* option. A negative value will force the task to run every time. Otherwise, a positive value implies the command should run when the number of loose objects is at least the value of *maintenance.loose-objects.auto*. The default value is 100.

maintenance.incremental-repack.auto

This integer config option controls how often the *incremental-repack* task should be run as part of *git maintenance run --auto*. If zero, then the *incremental-repack* task will not run with the *--auto* option. A negative value will force the task to run every time. Otherwise, a positive value implies the command should run when the number of pack-files not in the multi-pack-index is at least the value of *maintenance.incremental-repack.auto*. The default value is 10.

man.viewer

Specify the programs that may be used to display help in the *man* format. See Section G.3.63, “git-help(1)”.

man.<tool>.cmd

Specify the command to invoke the specified man viewer. The specified command is evaluated in shell with the man page passed as an argument. (See Section G.3.63, “git-help(1)”.)

man.<tool>.path

Override the path for the given tool that may be used to display help in the *man* format. See Section G.3.63, “git-help(1)”.

merge.conflictStyle

Specify the style in which conflicted hunks are written out to working tree files upon merge. The default is "merge", which shows a <<<<<<< conflict marker, changes made by one side, a ======= marker, changes made by the other side, and then a >>>>>>> marker. An alternate style, "diff3", adds a ||||||| marker and the original text before the ======= marker. The "merge" style tends to produce smaller conflict regions than diff3, both because of the exclusion of the original text, and because when a subset of lines match on the two sides, they are just pulled out of the conflict region. Another alternate style, "zdiff3", is similar to diff3 but removes matching lines on the two sides from the conflict region when those matching lines appear near either the beginning or end of a conflict region.

merge.defaultToUpstream

If merge is called without any commit argument, merge the upstream branches configured for the current branch by using their last observed values stored in their remote-tracking branches. The values of the *branch.<current branch>.merge* that name the branches at the remote named by *branch.<current branch>.remote* are consulted, and then they are mapped via *remote.<remote>.fetch* to their corresponding remote-tracking branches, and the tips of these tracking branches are merged. Defaults to true.

merge.ff

By default, Git does not create an extra merge commit when merging a commit that is a descendant of the current commit. Instead, the tip of the current branch is fast-forwarded. When set to *false*, this variable tells Git to create an extra merge commit in such a case (equivalent to giving the *--no-ff* option from the command line). When set to *only*, only such fast-forward merges are allowed (equivalent to giving the *--ff-only* option from the command line).

merge.verifySignatures

If true, this is equivalent to the --verify-signatures command line option. See Section G.3.86, “git-merge(1)” for details.

merge.branchdesc

In addition to branch names, populate the log message with the branch description text associated with them. Defaults to false.

merge.log

In addition to branch names, populate the log message with at most the specified number of one-line descriptions from the actual commits that are being merged. Defaults to false, and true is a synonym for 20.

merge.suppressDest

By adding a glob that matches the names of integration branches to this multi-valued configuration variable, the default merge message computed for merges into these integration branches will omit "into <branch name>" from its title.

An element with an empty value can be used to clear the list of globs accumulated from previous configuration entries. When there is no *merge.suppressDest* variable defined, the default value of *master* is used for backward compatibility.

merge.renameLimit

The number of files to consider in the exhaustive portion of rename detection during a merge. If not specified, defaults to the value of diff.renameLimit. If neither merge.renameLimit nor diff.renameLimit are specified, currently defaults to 7000. This setting has no effect if rename detection is turned off.

merge.renames

Whether Git detects renames. If set to "false", rename detection is disabled. If set to "true", basic rename detection is enabled. Defaults to the value of diff.renames.

merge.directoryRenames

Whether Git detects directory renames, affecting what happens at merge time to new files added to a directory on one side of history when that directory was renamed on the other side of history. If merge.directoryRenames is set to "false", directory rename detection is disabled, meaning that such new files will be left behind in the old directory. If set to "true", directory rename detection is enabled, meaning that such new files will be moved into the new directory. If set to "conflict", a conflict will be reported for such paths. If merge.renames is false, merge.directoryRenames is ignored and treated as false. Defaults to "conflict".

merge.renormalize

Tell Git that canonical representation of files in the repository has changed over time (e.g. earlier commits record text files with CRLF line endings, but recent ones use LF line endings). In such a repository, Git can convert the data recorded in commits to a canonical form before performing a merge to reduce unnecessary conflicts. For more information, see section "Merging branches with differing checkin/checkout attributes" in Section G.4.2, “gitattributes(5)”.

merge.stat

Whether to print the diffstat between ORIG_HEAD and the merge result at the end of the merge. True by default.

merge.autoStash

When set to true, automatically create a temporary stash entry before the operation begins, and apply it after the operation ends. This means that you can run merge on a dirty worktree. However, use with care: the final stash application after a successful merge might result in non-trivial conflicts. This option can be overridden by the *--no-autostash* and *--autostash* options of Section G.3.86, “git-merge(1)”. Defaults to false.

merge.tool

Controls which merge tool is used by Section G.3.88, “git-mergetool(1)”. The list below shows the valid built-in values. Any other value is treated as a custom merge tool and requires that a corresponding mergetool.<tool>.cmd variable is defined.

merge.guitool

Controls which merge tool is used by Section G.3.88, “git-mergetool(1)” when the -g/--gui flag is specified. The list below shows the valid built-in values. Any other value is treated as a custom merge tool and requires that a corresponding mergetool.<guitool>.cmd variable is defined.

araxis

Use Araxis Merge (requires a graphical session)

bc

Use Beyond Compare (requires a graphical session)

bc3

Use Beyond Compare (requires a graphical session)

bc4

Use Beyond Compare (requires a graphical session)

codecompare

Use Code Compare (requires a graphical session)

deltawalker

Use DeltaWalker (requires a graphical session)

diffmerge

Use DiffMerge (requires a graphical session)

diffuse

Use Diffuse (requires a graphical session)

ecmerge

Use ECMerge (requires a graphical session)

emerge

Use Emacs' Emerge

examdiff

Use ExamDiff Pro (requires a graphical session)

guiffy

Use Guiffy's Diff Tool (requires a graphical session)

gvimdiff

Use gVim (requires a graphical session) with a custom layout (see *git help mergetool*'s *BACKEND SPECIFIC HINTS* section)

gvimdiff1

Use gVim (requires a graphical session) with a 2 panes layout (LOCAL and REMOTE)

gvimdiff2

Use gVim (requires a graphical session) with a 3 panes layout (LOCAL, MERGED and REMOTE)

gvimdiff3

Use gVim (requires a graphical session) where only the MERGED file is shown

kdiff3

Use KDiff3 (requires a graphical session)

meld

Use Meld (requires a graphical session) with optional *auto merge* (see *git help mergetool*'s *CONFIGURATION* section)

nvimdiff

Use Neovim with a custom layout (see *git help mergetool*'s *BACKEND SPECIFIC HINTS* section)

nvimdiff1

Use Neovim with a 2 panes layout (LOCAL and REMOTE)

nvimdiff2

Use Neovim with a 3 panes layout (LOCAL, MERGED and REMOTE)

nvimdiff3

Use Neovim where only the MERGED file is shown

opendiff

Use FileMerge (requires a graphical session)

p4merge

Use HelixCore P4Merge (requires a graphical session)

smerge

Use Sublime Merge (requires a graphical session)

tkdiff

Use TkDiff (requires a graphical session)

tortoisemerge

Use TortoiseMerge (requires a graphical session)

vimdiff

Use Vim with a custom layout (see *git help mergetool*'s *BACKEND SPECIFIC HINTS* section)

vimdiff1

Use Vim with a 2 panes layout (LOCAL and REMOTE)

vimdiff2

Use Vim with a 3 panes layout (LOCAL, MERGED and REMOTE)

vimdiff3

Use Vim where only the MERGED file is shown

winmerge

Use WinMerge (requires a graphical session)

xxdiff

Use xxdiff (requires a graphical session)

merge.verbosity

Controls the amount of output shown by the recursive merge strategy. Level 0 outputs nothing except a final error message if conflicts were detected. Level 1 outputs only conflicts, 2 outputs conflicts and file changes. Level 5 and above outputs debugging information. The default is level 2. Can be overridden by the *GIT_MERGE_VERBOSITY* environment variable.

merge.<driver>.name

Defines a human-readable name for a custom low-level merge driver. See Section G.4.2, “gitattributes(5)” for details.

merge.<driver>.driver

Defines the command that implements a custom low-level merge driver. See Section G.4.2, “gitattributes(5)” for details.

merge.<driver>.recursive

Names a low-level merge driver to be used when performing an internal merge between common ancestors. See Section G.4.2, “gitattributes(5)” for details.

mergetool.<tool>.path

Override the path for the given tool. This is useful in case your tool is not in the PATH.

mergetool.<tool>.cmd

Specify the command to invoke the specified merge tool. The specified command is evaluated in shell with the following variables available: *BASE* is the name of a temporary file containing the common base of the files to be merged, if available; *LOCAL* is the name of a temporary file containing the contents of the file on the current branch; *REMOTE* is the name of a temporary file containing the contents of the file from the branch being merged; *MERGED* contains the name of the file to which the merge tool should write the results of a successful merge.

mergetool.<tool>.hideResolved

Allows the user to override the global *mergetool.hideResolved* value for a specific tool. See *mergetool.hideResolved* for the full description.

mergetool.<tool>.trustExitCode

For a custom merge command, specify whether the exit code of the merge command can be used to determine whether the merge was successful. If this is not set to true then the merge target file timestamp is checked, and the merge is assumed to have been successful if the file has been updated; otherwise, the user is prompted to indicate the success of the merge.

mergetool.meld.hasOutput

Older versions of *meld* do not support the *--output* option. Git will attempt to detect whether *meld* supports *--output* by inspecting the output of *meld --help*. Configuring *mergetool.meld.hasOutput* will make Git skip these checks and use the configured value instead. Setting *mergetool.meld.hasOutput* to *true* tells Git to unconditionally use the *--output* option, and *false* avoids using *--output*.

mergetool.meld.useAutoMerge

When the *--auto-merge* is given, meld will merge all non-conflicting parts automatically, highlight the conflicting parts, and wait for user decision. Setting *mergetool.meld.useAutoMerge* to *true* tells Git to unconditionally use the *--auto-merge* option with *meld*. Setting this value to *auto* makes git detect whether *--auto-merge* is supported and will only use *--auto-merge* when available. A value of *false* avoids using *--auto-merge* altogether, and is the default value.

mergetool.<vimdiff variant>.layout

Configure the split window layout for vimdiff's *<variant>*, which is any of *vimdiff*, *nvimdiff*, *gvimdiff*. Upon launching *git mergetool* with *--tool=<variant>* (or without *--tool* if *merge.tool* is configured as *<variant>*), Git will consult *mergetool.<variant>.layout* to determine the tool's layout. If the variant-specific configuration is not available, *vimdiff*'s is used as fallback. If that too is not available, a default layout with 4 windows will be used. To configure the layout, see the *BACKEND SPECIFIC HINTS* section in Section G.3.88, “git-mergetool(1)”.

mergetool.hideResolved

During a merge, Git will automatically resolve as many conflicts as possible and write the *MERGED* file containing conflict markers around any conflicts that it cannot resolve; *LOCAL* and *REMOTE* normally represent the versions of the file from before Git's conflict resolution. This flag causes *LOCAL* and *REMOTE* to be overwritten so that only the unresolved conflicts are presented to the merge tool. Can be configured per-tool via the *mergetool.<tool>.hideResolved* configuration variable. Defaults to *false*.

mergetool.keepBackup

After performing a merge, the original file with conflict markers can be saved as a file with a *.orig* extension. If this variable is set to *false* then this file is not preserved. Defaults to *true* (i.e. keep the backup files).

mergetool.keepTemporaries

When invoking a custom merge tool, Git uses a set of temporary files to pass to the tool. If the tool returns an error and this variable is set to *true*, then these temporary files will be preserved; otherwise, they will be removed after the tool has exited. Defaults to *false*.

mergetool.writeToTemp

Git writes temporary *BASE*, *LOCAL*, and *REMOTE* versions of conflicting files in the worktree by default. Git will attempt to use a temporary directory for these files when set *true*. Defaults to *false*.

mergetool.prompt

Prompt before each invocation of the merge resolution program.

mergetool.guiDefault

Set *true* to use the *merge.guitool* by default (equivalent to specifying the *--gui* argument), or *auto* to select *merge.guitool* or *merge.tool* depending on the presence of a *DISPLAY* environment variable value. The default is *false*, where the *--gui* argument must be provided explicitly for the *merge.guitool* to be used.

notes.mergeStrategy

Which merge strategy to choose by default when resolving notes conflicts. Must be one of *manual*, *ours*, *theirs*, *union*, or *cat_sort_uniq*. Defaults to *manual*. See the "NOTES MERGE STRATEGIES" section of Section G.3.94, “git-notes(1)” for more information on each strategy.

This setting can be overridden by passing the *--strategy* option to Section G.3.94, “git-notes(1)”.

notes.<name>.mergeStrategy

Which merge strategy to choose when doing a notes merge into refs/notes/<name>. This overrides the more general "notes.mergeStrategy". See the "NOTES MERGE STRATEGIES" section in Section G.3.94, “git-notes(1)” for more information on the available strategies.

notes.displayRef

Which ref (or refs, if a glob or specified more than once), in addition to the default set by *core.notesRef* or *GIT_NOTES_REF*, to read notes from when showing commit messages with the *git log* family of commands.

This setting can be overridden with the *GIT_NOTES_DISPLAY_REF* environment variable, which must be a colon separated list of refs or globs.

A warning will be issued for refs that do not exist, but a glob that does not match any refs is silently ignored.

This setting can be disabled by the *--no-notes* option to the *git log* family of commands, or by the *--notes=<ref>* option accepted by those commands.

The effective value of "core.notesRef" (possibly overridden by GIT_NOTES_REF) is also implicitly added to the list of refs to be displayed.

notes.rewrite.<command>

When rewriting commits with <command> (currently *amend* or *rebase*), if this variable is *false*, git will not copy notes from the original to the rewritten commit. Defaults to *true*. See also "*notes.rewriteRef*" below.

This setting can be overridden with the *GIT_NOTES_REWRITE_REF* environment variable, which must be a colon separated list of refs or globs.

notes.rewriteMode

When copying notes during a rewrite (see the "notes.rewrite.<command>" option), determines what to do if the target commit already has a note. Must be one of *overwrite*, *concatenate*, *cat_sort_uniq*, or *ignore*. Defaults to *concatenate*.

This setting can be overridden with the *GIT_NOTES_REWRITE_MODE* environment variable.

notes.rewriteRef

When copying notes during a rewrite, specifies the (fully qualified) ref whose notes should be copied. May be a glob, in which case notes in all matching refs will be copied. You may also specify this configuration several times.

Does not have a default value; you must configure this variable to enable note rewriting. Set it to *refs/notes/commits* to enable rewriting for the default commit notes.

Can be overridden with the *GIT_NOTES_REWRITE_REF* environment variable. See *notes.rewrite.<command>* above for a further description of its format.

pack.window

The size of the window used by Section G.3.96, “git-pack-objects(1)” when no window size is given on the command line. Defaults to 10.

pack.depth

The maximum delta depth used by Section G.3.96, “git-pack-objects(1)” when no maximum depth is given on the command line. Defaults to 50. Maximum value is 4095.

pack.windowMemory

The maximum size of memory that is consumed by each thread in Section G.3.96, “git-pack-objects(1)” for pack window memory when no limit is given on the command line. The value can be suffixed with "k", "m", or "g". When left unconfigured (or set explicitly to 0), there will be no limit.

pack.compression

An integer -1..9, indicating the compression level for objects in a pack file. -1 is the zlib default. 0 means no compression, and 1..9 are various speed/size tradeoffs, 9 being slowest. If not set, defaults to core.compression. If that is not set, defaults to -1, the zlib default, which is "a default compromise between speed and compression (currently equivalent to level 6)."

Note that changing the compression level will not automatically recompress all existing objects. You can force recompression by passing the -F option to Section G.3.113, “git-repack(1)”.

pack.allowPackReuse

When true or "single", and when reachability bitmaps are enabled, pack-objects will try to send parts of the bitmapped packfile verbatim. When "multi", and when a multi-pack reachability bitmap is available, pack-objects will try to send parts of all packs in the MIDX.

If only a single pack bitmap is available, and *pack.allowPackReuse* is set to "multi", reuse parts of just the bitmapped packfile. This can reduce memory and CPU usage to serve fetches, but might result in sending a slightly larger pack. Defaults to true.

pack.island

An extended regular expression configuring a set of delta islands. See "DELTA ISLANDS" in Section G.3.96, “git-pack-objects(1)” for details.

pack.islandCore

Specify an island name which gets to have its objects be packed first. This creates a kind of pseudo-pack at the front of one pack, so that the objects from the specified island are hopefully faster to copy into any pack that should be served to a user requesting these objects. In practice this means that the island specified

should likely correspond to what is the most commonly cloned in the repo. See also "DELTA ISLANDS" in Section G.3.96, “git-pack-objects(1)”.

pack.deltaCacheSize

The maximum memory in bytes used for caching deltas in Section G.3.96, “git-pack-objects(1)” before writing them out to a pack. This cache is used to speed up the writing object phase by not having to recompute the final delta result once the best match for all objects is found. Repacking large repositories on machines which are tight with memory might be badly impacted by this though, especially if this cache pushes the system into swapping. A value of 0 means no limit. The smallest size of 1 byte may be used to virtually disable this cache. Defaults to 256 MiB.

pack.deltaCacheLimit

The maximum size of a delta, that is cached in Section G.3.96, “git-pack-objects(1)”. This cache is used to speed up the writing object phase by not having to recompute the final delta result once the best match for all objects is found. Defaults to 1000. Maximum value is 65535.

pack.threads

Specifies the number of threads to spawn when searching for best delta matches. This requires that Section G.3.96, “git-pack-objects(1)” be compiled with pthreads otherwise this option is ignored with a warning. This is meant to reduce packing time on multiprocessor machines. The required amount of memory for the delta search window is however multiplied by the number of threads. Specifying 0 will cause Git to auto-detect the number of CPUs and set the number of threads accordingly.

pack.indexVersion

Specify the default pack index version. Valid values are 1 for legacy pack index used by Git versions prior to 1.5.2, and 2 for the new pack index with capabilities for packs larger than 4 GB as well as proper protection against the repacking of corrupted packs. Version 2 is the default. Note that version 2 is enforced and this config option is ignored whenever the corresponding pack is larger than 2 GB.

If you have an old Git that does not understand the version 2 *.*idx* file, cloning or fetching over a non-native protocol (e.g. "http") that will copy both *.*pack* file and corresponding *.*idx* file from the other side may give you a repository that cannot be accessed with your older version of Git. If the *.*pack* file is smaller than 2 GB, however, you can use Section G.3.69, “git-index-pack(1)” on the *.pack file to regenerate the *.*idx* file.

pack.packSizeLimit

The maximum size of a pack. This setting only affects packing to a file when repacking, i.e. the git:// protocol is unaffected. It can be overridden by the *--max-pack-size* option of Section G.3.113, “git-repack(1)”. Reaching this limit results in the creation of multiple packfiles.

Note that this option is rarely useful, and may result in a larger total on-disk size (because Git will not store deltas between packs) and worse runtime performance (object lookup within multiple packs is slower than a single pack, and optimizations like reachability bitmaps cannot cope with multiple packs).

If you need to actively run Git using smaller packfiles (e.g., because your filesystem does not support large files), this option may help. But if your goal is to transmit a packfile over a medium that supports limited sizes (e.g., removable media that cannot store the whole repository), you are likely better off creating a single large packfile and splitting it using a generic multi-volume archive tool (e.g., Unix *split*).

The minimum size allowed is limited to 1 MiB. The default is unlimited. Common unit suffixes of *k*, *m*, or *g* are supported.

pack.useBitmaps

When true, git will use pack bitmaps (if available) when packing to stdout (e.g., during the server side of a fetch). Defaults to true. You should not generally need to turn this off unless you are debugging pack bitmaps.

pack.useBitmapBoundaryTraversal

When true, Git will use an experimental algorithm for computing reachability queries with bitmaps. Instead of building up complete bitmaps for all of the negated tips and then OR-ing them together, consider negated tips with existing bitmaps as additive (i.e. OR-ing them into the result if they exist, ignoring them otherwise), and build up a bitmap at the boundary instead.

When using this algorithm, Git may include too many objects as a result of not opening up trees belonging to certain UNINTERESTING commits. This inexactness matches the non-bitmap traversal algorithm.

In many cases, this can provide a speed-up over the exact algorithm, particularly when there is poor bitmap coverage of the negated side of the query.

pack.useSparse

When true, git will default to using the *--sparse* option in *git pack-objects* when the *--revs* option is present. This algorithm only walks trees that appear in paths that introduce new objects. This can have significant performance benefits when computing a pack to send a small change. However, it is possible that extra objects are added to the pack-file if the included commits contain certain types of direct renames. Default is *true*.

pack.preferBitmapTips

When selecting which commits will receive bitmaps, prefer a commit at the tip of any reference that is a suffix of any value of this configuration over any other commits in the "selection window".

Note that setting this configuration to *refs/foo* does not mean that the commits at the tips of *refs/foo/bar* and *refs/foo/baz* will necessarily be selected. This is because commits are selected for bitmaps from within a series of windows of variable length.

If a commit at the tip of any reference which is a suffix of any value of this configuration is seen in a window, it is immediately given preference over any other commit in that window.

pack.writeBitmaps (deprecated)

This is a deprecated synonym for *repack.writeBitmaps*.

pack.writeBitmapHashCache

When true, git will include a "hash cache" section in the bitmap index (if one is written). This cache can be used to feed git's delta heuristics, potentially leading to better deltas between bitmapped and non-bitmapped objects (e.g., when serving a fetch between an older, bitmapped pack and objects that have been pushed since the last gc). The downside is that it consumes 4 bytes per object of disk space. Defaults to true.

When writing a multi-pack reachability bitmap, no new namehashes are computed; instead, any namehashes stored in an existing bitmap are permuted into their appropriate location when writing a new bitmap.

pack.writeBitmapLookupTable

When true, Git will include a "lookup table" section in the bitmap index (if one is written). This table is used to defer loading individual bitmaps as late as possible. This can be beneficial in repositories that have relatively large bitmap indexes. Defaults to false.

pack.readReverseIndex

When true, git will read any .rev file(s) that may be available (see: Section G.5.5, “gitformat-pack(5)”). When false, the reverse index will be generated from scratch and stored in memory. Defaults to true.

pack.writeReverseIndex

When true, git will write a corresponding .rev file (see: Section G.5.5, “gitformat-pack(5)”) for each new packfile that it writes in all places except for Section G.3.47, “git-fast-import(1)” and in the bulk checkin mechanism. Defaults to true.

pager.<cmd>

If the value is boolean, turns on or off pagination of the output of a particular Git subcommand when writing to a tty. Otherwise, turns on pagination for the subcommand using the pager specified by the value of *pager.<cmd>*. If *--paginate* or *--no-pager* is specified on the command line, it takes precedence over this option. To disable pagination for all commands, set *core.pager* or *GIT_PAGER* to *cat*.

pretty.<name>

Alias for a --pretty= format string, as specified in Section G.3.74, “git-log(1)”. Any aliases defined here can be used just as the built-in pretty formats could. For example, running *git config pretty.changelog "format:* %H %s"* would cause the invocation *git log --pretty=changelog* to be equivalent to running *git log "--pretty=format:* %H %s"*. Note that an alias with the same name as a built-in format will be silently ignored.

protocol.allow

If set, provide a user defined default policy for all protocols which don't explicitly have a policy (*protocol.<name>.allow*). By default, if unset, known-safe protocols (http, https, git, ssh) have a default policy of *always*, known-dangerous protocols (ext) have a default policy of *never*, and all other protocols (including file) have a default policy of *user*. Supported policies:

- *always* - protocol is always able to be used.
- *never* - protocol is never able to be used.
- *user* - protocol is only able to be used when *GIT_PROTOCOL_FROM_USER* is either unset or has a value of 1. This policy should be used when you want a protocol to be directly usable by the user but don't want it used by commands which execute clone/fetch/push commands without user input, e.g. recursive submodule initialization.

protocol.<name>.allow

Set a policy to be used by protocol *<name>* with clone/fetch/push commands. See *protocol.allow* above for the available policies.

The protocol names currently used by git are:

- *file*: any local file-based path (including *file:*// URLs, or local paths)
- *git*: the anonymous git protocol over a direct TCP connection (or proxy, if configured)
- *ssh*: git over ssh (including *host:path* syntax, *ssh://*, etc).
- *http*: git over http, both "smart http" and "dumb http". Note that this does *not* include *https*; if you want to configure both, you must do so individually.
- any external helpers are named by their protocol (e.g., use *hg* to allow the *git-remote-hg* helper)

protocol.version

If set, clients will attempt to communicate with a server using the specified protocol version. If the server does not support it, communication falls back to version 0. If unset, the default is *2*. Supported versions:

- *0* - the original wire protocol.
- *1* - the original wire protocol with the addition of a version string in the initial response from the server.
- *2* - Wire protocol version 2, see Section G.5.11, “gitprotocol-v2(5)”.

pull.ff

By default, Git does not create an extra merge commit when merging a commit that is a descendant of the current commit. Instead, the tip of the current branch is fast-forwarded. When set to *false*, this variable tells

Git to create an extra merge commit in such a case (equivalent to giving the *--no-ff* option from the command line). When set to *only*, only such fast-forward merges are allowed (equivalent to giving the *--ff-only* option from the command line). This setting overrides *merge.ff* when pulling.

pull.rebase

When true, rebase branches on top of the fetched branch, instead of merging the default branch from the default remote when "git pull" is run. See "branch.<name>.rebase" for setting this on a per-branch basis.

When *merges* (or just *m*), pass the *--rebase-merges* option to *git rebase* so that the local merge commits are included in the rebase (see Section G.3.107, “git-rebase(1)” for details).

When the value is *interactive* (or just *i*), the rebase is run in interactive mode.

NOTE: this is a possibly dangerous operation; do **not** use it unless you understand the implications (see Section G.3.107, “git-rebase(1)” for details).

pull.octopus

The default merge strategy to use when pulling multiple branches at once.

pull.twohead

The default merge strategy to use when pulling a single branch.

push.autoSetupRemote

If set to "true" assume *--set-upstream* on default push when no upstream tracking exists for the current branch; this option takes effect with push.default options *simple*, *upstream*, and *current*. It is useful if by default you want new branches to be pushed to the default remote (like the behavior of *push.default=current*) and you also want the upstream tracking to be set. Workflows most likely to benefit from this option are *simple* central workflows where all branches are expected to have the same name on the remote.

push.default

Defines the action *git push* should take if no refspec is given (whether from the command-line, config, or elsewhere). Different values are well-suited for specific workflows; for instance, in a purely central workflow (i.e. the fetch source is equal to the push destination), *upstream* is probably what you want. Possible values are:

- *nothing* - do not push anything (error out) unless a refspec is given. This is primarily meant for people who want to avoid mistakes by always being explicit.
- *current* - push the current branch to update a branch with the same name on the receiving end. Works in both central and non-central workflows.
- *upstream* - push the current branch back to the branch whose changes are usually integrated into the current branch (which is called *@{upstream}*). This mode only makes sense if you are pushing to the same repository you would normally pull from (i.e. central workflow).
- *tracking* - This is a deprecated synonym for *upstream*.
- *simple* - push the current branch with the same name on the remote.

  If you are working on a centralized workflow (pushing to the same repository you pull from, which is typically *origin*), then you need to configure an upstream branch with the same name.

  This mode is the default since Git 2.0, and is the safest option suited for beginners.
- *matching* - push all branches having the same name on both ends. This makes the repository you are pushing to remember the set of branches that will be pushed out (e.g. if you always push *maint* and *master* there and no other branches, the repository you push to will have these two branches, and your local *maint* and *master* will be pushed there).

To use this mode effectively, you have to make sure *all* the branches you would push out are ready to be pushed out before running *git push*, as the whole point of this mode is to allow you to push all of the branches in one go. If you usually finish work on only one branch and push out the result, while other branches are unfinished, this mode is not for you. Also this mode is not suitable for pushing into a shared central repository, as other people may add new branches there, or update the tip of existing branches outside your control.

This used to be the default, but not since Git 2.0 (*simple* is the new default).

push.followTags

If set to true, enable *--follow-tags* option by default. You may override this configuration at time of push by specifying *--no-follow-tags*.

push.gpgSign

May be set to a boolean value, or the string *if-asked*. A true value causes all pushes to be GPG signed, as if *--signed* is passed to Section G.3.103, “git-push(1)”. The string *if-asked* causes pushes to be signed if the server supports it, as if *--signed=if-asked* is passed to *git push*. A false value may override a value from a lower-priority config file. An explicit command-line flag always overrides this config option.

push.pushOption

When no *--push-option=<option>* argument is given from the command line, *git push* behaves as if each <value> of this variable is given as *--push-option=<value>*.

This is a multi-valued variable, and an empty value can be used in a higher priority configuration file (e.g. *.git/config* in a repository) to clear the values inherited from a lower priority configuration files (e.g. *$HOME/.gitconfig*).

```
Example:

/etc/gitconfig
  push.pushoption = a
  push.pushoption = b

~/.gitconfig
  push.pushoption = c

repo/.git/config
  push.pushoption =
  push.pushoption = b

This will result in only b (a and c are cleared).
```

push.recurseSubmodules

May be "check", "on-demand", "only", or "no", with the same behavior as that of "push --recurse-submodules". If not set, *no* is used by default, unless *submodule.recurse* is set (in which case a *true* value means *on-demand*).

push.useForceIfIncludes

If set to "true", it is equivalent to specifying *--force-if-includes* as an option to Section G.3.103, “git-push(1)” in the command line. Adding *--no-force-if-includes* at the time of push overrides this configuration setting.

push.negotiate

If set to "true", attempt to reduce the size of the packfile sent by rounds of negotiation in which the client and the server attempt to find commits in common. If "false", Git will rely solely on the server's ref advertisement to find commits in common.

push.useBitmaps

If set to "false", disable use of bitmaps for "git push" even if *pack.useBitmaps* is "true", without preventing other git operations from using bitmaps. Default is true.

rebase.backend

Default backend to use for rebasing. Possible choices are *apply* or *merge*. In the future, if the merge backend gains all remaining capabilities of the apply backend, this setting may become unused.

rebase.stat

Whether to show a diffstat of what changed upstream since the last rebase. False by default.

rebase.autoSquash

If set to true, enable the *--autosquash* option of Section G.3.107, “git-rebase(1)” by default for interactive mode. This can be overridden with the *--no-autosquash* option.

rebase.autoStash

When set to true, automatically create a temporary stash entry before the operation begins, and apply it after the operation ends. This means that you can run rebase on a dirty worktree. However, use with care: the final stash application after a successful rebase might result in non-trivial conflicts. This option can be overridden by the *--no-autostash* and *--autostash* options of Section G.3.107, “git-rebase(1)”. Defaults to false.

rebase.updateRefs

If set to true enable *--update-refs* option by default.

rebase.missingCommitsCheck

If set to "warn", git rebase -i will print a warning if some commits are removed (e.g. a line was deleted), however the rebase will still proceed. If set to "error", it will print the previous warning and stop the rebase, *git rebase --edit-todo* can then be used to correct the error. If set to "ignore", no checking is done. To drop a commit without warning or error, use the *drop* command in the todo list. Defaults to "ignore".

rebase.instructionFormat

A format string, as specified in Section G.3.74, “git-log(1)”, to be used for the todo list during an interactive rebase. The format will automatically have the commit hash prepended to the format.

rebase.abbreviateCommands

If set to true, *git rebase* will use abbreviated command names in the todo list resulting in something like this:

```
p deadbee The oneline of the commit
p fa1afe1 The oneline of the next commit
...
```

instead of:

```
pick deadbee The oneline of the commit
pick fa1afe1 The oneline of the next commit
...
```

Defaults to false.

rebase.rescheduleFailedExec

Automatically reschedule *exec* commands that failed. This only makes sense in interactive mode (or when an *--exec* option was provided). This is the same as specifying the *--reschedule-failed-exec* option.

rebase.forkPoint

If set to false set *--no-fork-point* option by default.

rebase.rebaseMerges

Whether and how to set the *--rebase-merges* option by default. Can be *rebase-cousins*, *no-rebase-cousins*, or a boolean. Setting to true or to *no-rebase-cousins* is equivalent to *--rebase-merges=no-rebase-cousins*, setting to *rebase-cousins* is equivalent to *--rebase-merges=rebase-cousins*, and setting to false is equivalent to *--no-rebase-merges*. Passing *--rebase-merges* on the command line, with or without an argument, overrides any *rebase.rebaseMerges* configuration.

rebase.maxLabelLength

When generating label names from commit subjects, truncate the names to this length. By default, the names are truncated to a little less than *NAME_MAX* (to allow e.g. *.lock* files to be written for the corresponding loose refs).

receive.advertiseAtomic

By default, git-receive-pack will advertise the atomic push capability to its clients. If you don't want to advertise this capability, set this variable to false.

receive.advertisePushOptions

When set to true, git-receive-pack will advertise the push options capability to its clients. False by default.

receive.autogc

By default, git-receive-pack will run "git maintenance run --auto" after receiving data from git-push and updating refs. You can stop it by setting this variable to false.

receive.certNonceSeed

By setting this variable to a string, *git receive-pack* will accept a *git push --signed* and verify it by using a "nonce" protected by HMAC using this string as a secret key.

receive.certNonceSlop

When a *git push --signed* sends a push certificate with a "nonce" that was issued by a receive-pack serving the same repository within this many seconds, export the "nonce" found in the certificate to *GIT_PUSH_CERT_NONCE* to the hooks (instead of what the receive-pack asked the sending side to include). This may allow writing checks in *pre-receive* and *post-receive* a bit easier. Instead of checking *GIT_PUSH_CERT_NONCE_SLOP* environment variable that records by how many seconds the nonce is stale to decide if they want to accept the certificate, they only can check *GIT_PUSH_CERT_NONCE_STATUS* is *OK*.

receive.fsckObjects

If it is set to true, git-receive-pack will check all received objects. See *transfer.fsckObjects* for what's checked. Defaults to false. If not set, the value of *transfer.fsckObjects* is used instead.

receive.fsck.<msg-id>

Acts like *fsck.<msg-id>*, but is used by Section G.3.108, “git-receive-pack(1)” instead of Section G.3.56, “git-fsck(1)”. See the *fsck.<msg-id>* documentation for details.

receive.fsck.skipList

Acts like *fsck.skipList*, but is used by Section G.3.108, “git-receive-pack(1)” instead of Section G.3.56, “git-fsck(1)”. See the *fsck.skipList* documentation for details.

receive.keepAlive

After receiving the pack from the client, *receive-pack* may produce no output (if *--quiet* was specified) while processing the pack, causing some networks to drop the TCP connection. With this option set, if *receive-pack* does not transmit any data in this phase for *receive.keepAlive* seconds, it will send a short keepalive packet. The default is 5 seconds; set to 0 to disable keepalives entirely.

receive.unpackLimit

If the number of objects received in a push is below this limit then the objects will be unpacked into loose object files. However if the number of received objects equals or exceeds this limit then the received pack will be stored as a pack, after adding any missing delta bases. Storing the pack from a push can make the push operation complete faster, especially on slow filesystems. If not set, the value of *transfer.unpackLimit* is used instead.

receive.maxInputSize

If the size of the incoming pack stream is larger than this limit, then git-receive-pack will error out, instead of accepting the pack file. If not set or set to 0, then the size is unlimited.

receive.denyDeletes

If set to true, git-receive-pack will deny a ref update that deletes the ref. Use this to prevent such a ref deletion via a push.

receive.denyDeleteCurrent

If set to true, git-receive-pack will deny a ref update that deletes the currently checked out branch of a non-bare repository.

receive.denyCurrentBranch

If set to true or "refuse", git-receive-pack will deny a ref update to the currently checked out branch of a non-bare repository. Such a push is potentially dangerous because it brings the HEAD out of sync with the index and working tree. If set to "warn", print a warning of such a push to stderr, but allow the push to proceed. If set to false or "ignore", allow such pushes with no message. Defaults to "refuse".

Another option is "updateInstead" which will update the working tree if pushing into the current branch. This option is intended for synchronizing working directories when one side is not easily accessible via interactive ssh (e.g. a live web site, hence the requirement that the working directory be clean). This mode also comes in handy when developing inside a VM to test and fix code on different Operating Systems.

By default, "updateInstead" will refuse the push if the working tree or the index have any difference from the HEAD, but the *push-to-checkout* hook can be used to customize this. See Section G.4.7, “githooks(5)”.

receive.denyNonFastForwards

If set to true, git-receive-pack will deny a ref update which is not a fast-forward. Use this to prevent such an update via a push, even if that push is forced. This configuration variable is set when initializing a shared repository.

receive.hideRefs

This variable is the same as *transfer.hideRefs*, but applies only to *receive-pack* (and so affects pushes, but not fetches). An attempt to update or delete a hidden ref by *git push* is rejected.

receive.procReceiveRefs

This is a multi-valued variable that defines reference prefixes to match the commands in *receive-pack*. Commands matching the prefixes will be executed by an external hook "proc-receive", instead of the internal *ex-*

ecute_commands function. If this variable is not defined, the "proc-receive" hook will never be used, and all commands will be executed by the internal *execute_commands* function.

For example, if this variable is set to "refs/for", pushing to reference such as "refs/for/master" will not create or update a reference named "refs/for/master", but may create or update a pull request directly by running the hook "proc-receive".

Optional modifiers can be provided in the beginning of the value to filter commands for specific actions: create (a), modify (m), delete (d). A *!* can be included in the modifiers to negate the reference prefix entry. E.g.:

```
git config --system --add receive.procReceiveRefs ad:refs/heads
git config --system --add receive.procReceiveRefs !:refs/heads
```

receive.updateServerInfo

If set to true, git-receive-pack will run git-update-server-info after receiving data from git-push and updating refs.

receive.shallowUpdate

If set to true, .git/shallow can be updated when new refs require new shallow roots. Otherwise those refs are rejected.

remote.pushDefault

The remote to push to by default. Overrides *branch.<name>.remote* for all branches, and is overridden by *branch.<name>.pushRemote* for specific branches.

remote.<name>.url

The URL of a remote repository. See Section G.3.49, “git-fetch(1)” or Section G.3.103, “git-push(1)”.

remote.<name>.pushurl

The push URL of a remote repository. See Section G.3.103, “git-push(1)”.

remote.<name>.proxy

For remotes that require curl (http, https and ftp), the URL to the proxy to use for that remote. Set to the empty string to disable proxying for that remote.

remote.<name>.proxyAuthMethod

For remotes that require curl (http, https and ftp), the method to use for authenticating against the proxy in use (probably set in *remote.<name>.proxy*). See *http.proxyAuthMethod*.

remote.<name>.fetch

The default set of "refspec" for Section G.3.49, “git-fetch(1)”. See Section G.3.49, “git-fetch(1)”.

remote.<name>.push

The default set of "refspec" for Section G.3.103, “git-push(1)”. See Section G.3.103, “git-push(1)”.

remote.<name>.mirror

If true, pushing to this remote will automatically behave as if the *--mirror* option was given on the command line.

remote.<name>.skipDefaultUpdate

If true, this remote will be skipped by default when updating using Section G.3.49, “git-fetch(1)” or the *update* subcommand of Section G.3.112, “git-remote(1)”.

remote.<name>.skipFetchAll

If true, this remote will be skipped by default when updating using Section G.3.49, “git-fetch(1)” or the *update* subcommand of Section G.3.112, “git-remote(1)”.

remote.<name>.receivepack

The default program to execute on the remote side when pushing. See option --receive-pack of Section G.3.103, “git-push(1)”.

remote.<name>.uploadpack

The default program to execute on the remote side when fetching. See option --upload-pack of Section G.3.48, “git-fetch-pack(1)”.

remote.<name>.tagOpt

Setting this value to --no-tags disables automatic tag following when fetching from remote <name>. Setting it to --tags will fetch every tag from remote <name>, even if they are not reachable from remote branch heads. Passing these flags directly to Section G.3.49, “git-fetch(1)” can override this setting. See options --tags and --no-tags of Section G.3.49, “git-fetch(1)”.

remote.<name>.vcs

Setting this to a value <vcs> will cause Git to interact with the remote with the git-remote-<vcs> helper.

remote.<name>.prune

When set to true, fetching from this remote by default will also remove any remote-tracking references that no longer exist on the remote (as if the *--prune* option was given on the command line). Overrides *fetch.prune* settings, if any.

remote.<name>.pruneTags

When set to true, fetching from this remote by default will also remove any local tags that no longer exist on the remote if pruning is activated in general via *remote.<name>.prune*, *fetch.prune* or *--prune*. Overrides *fetch.pruneTags* settings, if any.

See also *remote.<name>.prune* and the PRUNING section of Section G.3.49, “git-fetch(1)”.

remote.<name>.promisor

When set to true, this remote will be used to fetch promisor objects.

remote.<name>.partialclonefilter

The filter that will be applied when fetching from this promisor remote. Changing or clearing this value will only affect fetches for new commits. To fetch associated objects for commits already present in the local object database, use the *--refetch* option of Section G.3.49, “git-fetch(1)”.

remotes.<group>

The list of remotes which are fetched by "git remote update <group>". See Section G.3.112, “git-remote(1)”.

repack.useDeltaBaseOffset

By default, Section G.3.113, “git-repack(1)” creates packs that use delta-base offset. If you need to share your repository with Git older than version 1.4.4, either directly or via a dumb protocol such as http, then you need to set this option to "false" and repack. Access from old Git versions over the native protocol are unaffected by this option.

repack.packKeptObjects

If set to true, makes *git repack* act as if *--pack-kept-objects* was passed. See Section G.3.113, “git-repack(1)” for details. Defaults to *false* normally, but *true* if a bitmap index is being written (either via *--write-bitmap-index* or *repack.writeBitmaps*).

repack.useDeltaIslands

If set to true, makes *git repack* act as if *--delta-islands* was passed. Defaults to *false*.

repack.writeBitmaps

When true, git will write a bitmap index when packing all objects to disk (e.g., when *git repack -a* is run). This index can speed up the "counting objects" phase of subsequent packs created for clones and fetches, at the cost of some disk space and extra time spent on the initial repack. This has no effect if multiple packfiles are created. Defaults to true on bare repos, false otherwise.

repack.updateServerInfo

If set to false, Section G.3.113, “git-repack(1)” will not run Section G.3.149, “git-update-server-info(1)”. Defaults to true. Can be overridden when true by the *-n* option of Section G.3.113, “git-repack(1)”.

repack.cruftWindow , repack.cruftWindowMemory , repack.cruftDepth , repack.cruftThreads

Parameters used by Section G.3.96, “git-pack-objects(1)” when generating a cruft pack and the respective parameters are not given over the command line. See similarly named *pack.** configuration variables for defaults and meaning.

rerere.autoUpdate

When set to true, *git-rerere* updates the index with the resulting contents after it cleanly resolves conflicts using previously recorded resolutions. Defaults to false.

rerere.enabled

Activate recording of resolved conflicts, so that identical conflict hunks can be resolved automatically, should they be encountered again. By default, Section G.3.117, “git-rerere(1)” is enabled if there is an *rr-cache* directory under the *$GIT_DIR*, e.g. if "rerere" was previously used in the repository.

revert.reference

Setting this variable to true makes *git revert* behave as if the *--reference* option is given.

safe.bareRepository

Specifies which bare repositories Git will work with. The currently supported values are:

- *all*: Git works with all bare repositories. This is the default.
- *explicit*: Git only works with bare repositories specified via the top-level *--git-dir* command-line option, or the *GIT_DIR* environment variable (see Section G.3.1, “git(1)”).

  If you do not use bare repositories in your workflow, then it may be beneficial to set *safe.bareRepository* to *explicit* in your global config. This will protect you from attacks that involve cloning a repository that contains a bare repository and running a Git command within that directory.

  This config setting is only respected in protected configuration (see the section called “SCOPES”). This prevents untrusted repositories from tampering with this value.

safe.directory

These config entries specify Git-tracked directories that are considered safe even if they are owned by someone other than the current user. By default, Git will refuse to even parse a Git config of a repository owned by

someone else, let alone run its hooks, and this config setting allows users to specify exceptions, e.g. for intentionally shared repositories (see the *--shared* option in Section G.3.71, “git-init(1)”).

This is a multi-valued setting, i.e. you can add more than one directory via *git config --add*. To reset the list of safe directories (e.g. to override any such directories specified in the system config), add a *safe.directory* entry with an empty value.

This config setting is only respected in protected configuration (see the section called “SCOPES”). This prevents untrusted repositories from tampering with this value.

The value of this setting is interpolated, i.e. *~/<path>* expands to a path relative to the home directory and *%(prefix)/<path>* expands to a path relative to Git's (runtime) prefix.

To completely opt-out of this security check, set *safe.directory* to the string *. This will allow all repositories to be treated as if their directory was listed in the *safe.directory* list. If *safe.directory=** is set in system config and you want to re-enable this protection, then initialize your list with an empty value before listing the repositories that you deem safe. Giving a directory with /* appended to it will allow access to all repositories under the named directory.

As explained, Git only allows you to access repositories owned by yourself, i.e. the user who is running Git, by default. When Git is running as *root* in a non Windows platform that provides sudo, however, git checks the SUDO_UID environment variable that sudo creates and will allow access to the uid recorded as its value in addition to the id from *root*. This is to make it easy to perform a common sequence during installation "make && sudo make install". A git process running under *sudo* runs as *root* but the *sudo* command exports the environment variable to record which id the original user has. If that is not what you would prefer and want git to only trust repositories that are owned by root instead, then you can remove the *SUDO_UID* variable from root's environment before invoking git.

sendemail.identity

A configuration identity. When given, causes values in the *sendemail.<identity>* subsection to take precedence over values in the *sendemail* section. The default identity is the value of *sendemail.identity*.

sendemail.smtpEncryption

See Section G.3.124, “git-send-email(1)” for description. Note that this setting is not subject to the *identity* mechanism.

sendemail.smtpSSLCertPath

Path to ca-certificates (either a directory or a single file). Set it to an empty string to disable certificate verification.

sendemail.<identity>.*

Identity-specific versions of the *sendemail.** parameters found below, taking precedence over those when this identity is selected, through either the command-line or *sendemail.identity*.

sendemail.multiEdit

If true (default), a single editor instance will be spawned to edit files you have to edit (patches when *--annotate* is used, and the summary when *--compose* is used). If false, files will be edited one after the other, spawning a new editor each time.

sendemail.confirm

Sets the default for whether to confirm before sending. Must be one of *always*, *never*, *cc*, *compose*, or *auto*. See *--confirm* in the Section G.3.124, “git-send-email(1)” documentation for the meaning of these values.

sendemail.aliasesFile

To avoid typing long email addresses, point this to one or more email aliases files. You must also supply *sendemail.aliasFileType*.

sendemail.aliasFileType

Format of the file(s) specified in sendemail.aliasesFile. Must be one of *mutt*, *mailrc*, *pine*, *elm*, *gnus*, or *sendmail*.

What an alias file in each format looks like can be found in the documentation of the email program of the same name. The differences and limitations from the standard formats are described below:

sendmail

- Quoted aliases and quoted addresses are not supported: lines that contain a " symbol are ignored.
- Redirection to a file (*/path/name*) or pipe (|*command*) is not supported.
- File inclusion (*:include: /path/name*) is not supported.
- Warnings are printed on the standard error output for any explicitly unsupported constructs, and any other lines that are not recognized by the parser.

sendemail.annotate , sendemail.bcc , sendemail.cc , sendemail.ccCmd , sendemail.chainReplyTo , sendemail.envelopeSender , sendemail.from , sendemail.headerCmd , sendemail.signedOffByCc , sendemail.smtpPass , sendemail.suppressCc , sendemail.suppressFrom , sendemail.to , sendemail.toCmd , sendemail.smtpDomain , sendemail.smtpServer , sendemail.smtpServerPort , sendemail.smtpServerOption , sendemail.smtpUser , sendemail.thread , sendemail.transferEncoding , sendemail.validate , sendemail.xmailer

These configuration variables all provide a default for Section G.3.124, “git-send-email(1)” command-line options. See its documentation for details.

sendemail.signedOffCc (deprecated)

Deprecated alias for *sendemail.signedOffByCc*.

sendemail.smtpBatchSize

Number of messages to be sent per connection, after that a relogin will happen. If the value is 0 or undefined, send all messages in one connection. See also the *--batch-size* option of Section G.3.124, “git-send-email(1)”.

sendemail.smtpReloginDelay

Seconds to wait before reconnecting to the smtp server. See also the *--relogin-delay* option of Section G.3.124, “git-send-email(1)”.

sendemail.forbidSendmailVariables

To avoid common misconfiguration mistakes, Section G.3.124, “git-send-email(1)” will abort with a warning if any configuration options for "sendmail" exist. Set this variable to bypass the check.

sequence.editor

Text editor used by *git rebase -i* for editing the rebase instruction file. The value is meant to be interpreted by the shell when it is used. It can be overridden by the *GIT_SEQUENCE_EDITOR* environment variable. When not configured, the default commit message editor is used instead.

showBranch.default

The default set of branches for Section G.3.131, “git-show-branch(1)”. See Section G.3.131, “git-show-branch(1)”.

sparse.expectFilesOutsideOfPatterns

Typically with sparse checkouts, files not matching any sparsity patterns are marked with a SKIP_WORKTREE bit in the index and are missing from the working tree. Accordingly, Git will ordinarily check whether files with the SKIP_WORKTREE bit are in fact present in the working tree contrary to expectations. If Git

finds any, it marks those paths as present by clearing the relevant SKIP_WORKTREE bits. This option can be used to tell Git that such present-despite-skipped files are expected and to stop checking for them.

The default is *false*, which allows Git to automatically recover from the list of files in the index and working tree falling out of sync.

Set this to *true* if you are in a setup where some external factor relieves Git of the responsibility for maintaining the consistency between the presence of working tree files and sparsity patterns. For example, if you have a Git-aware virtual file system that has a robust mechanism for keeping the working tree and the sparsity patterns up to date based on access patterns.

Regardless of this setting, Git does not check for present-despite-skipped files unless sparse checkout is enabled, so this config option has no effect unless *core.sparseCheckout* is *true*.

splitIndex.maxPercentChange

When the split index feature is used, this specifies the percent of entries the split index can contain compared to the total number of entries in both the split index and the shared index before a new shared index is written. The value should be between 0 and 100. If the value is 0, then a new shared index is always written; if it is 100, a new shared index is never written. By default, the value is 20, so a new shared index is written if the number of entries in the split index would be greater than 20 percent of the total number of entries. See Section G.3.147, “git-update-index(1)”.

splitIndex.sharedIndexExpire

When the split index feature is used, shared index files that were not modified since the time this variable specifies will be removed when a new shared index file is created. The value "now" expires all entries immediately, and "never" suppresses expiration altogether. The default value is "2.weeks.ago". Note that a shared index file is considered modified (for the purpose of expiration) each time a new split-index file is either created based on it or read from it. See Section G.3.147, “git-update-index(1)”.

ssh.variant

By default, Git determines the command line arguments to use based on the basename of the configured SSH command (configured using the environment variable *GIT_SSH* or *GIT_SSH_COMMAND* or the config setting *core.sshCommand*). If the basename is unrecognized, Git will attempt to detect support of OpenSSH options by first invoking the configured SSH command with the *-G* (print configuration) option and will subsequently use OpenSSH options (if that is successful) or no options besides the host and remote command (if it fails).

The config variable *ssh.variant* can be set to override this detection. Valid values are *ssh* (to use OpenSSH options), *plink*, *putty*, *tortoiseplink*, *simple* (no options except the host and remote command). The default auto-detection can be explicitly requested using the value *auto*. Any other value is treated as *ssh*. This setting can also be overridden via the environment variable *GIT_SSH_VARIANT*.

The current command-line parameters used for each variant are as follows:

- *ssh* - [-p port] [-4] [-6] [-o option] [username@]host command
- *simple* - [username@]host command
- *plink* or *putty* - [-P port] [-4] [-6] [username@]host command
- *tortoiseplink* - [-P port] [-4] [-6] -batch [username@]host command

Except for the *simple* variant, command-line parameters are likely to change as git gains new features.

stash.showIncludeUntracked

If this is set to true, the *git stash show* command will show the untracked files of a stash entry. Defaults to false. See the description of the *show* command in Section G.3.137, “git-stash(1)”.

stash.showPatch

If this is set to true, the *git stash show* command without an option will show the stash entry in patch form. Defaults to false. See the description of the *show* command in Section G.3.137, “git-stash(1)”.

stash.showStat

If this is set to true, the *git stash show* command without an option will show a diffstat of the stash entry. Defaults to true. See the description of the *show* command in Section G.3.137, “git-stash(1)”.

status.relativePaths

By default, Section G.3.138, “git-status(1)” shows paths relative to the current directory. Setting this variable to *false* shows paths relative to the repository root (this was the default for Git prior to v1.5.4).

status.short

Set to true to enable --short by default in Section G.3.138, “git-status(1)”. The option --no-short takes precedence over this variable.

status.branch

Set to true to enable --branch by default in Section G.3.138, “git-status(1)”. The option --no-branch takes precedence over this variable.

status.aheadBehind

Set to true to enable *--ahead-behind* and false to enable *--no-ahead-behind* by default in Section G.3.138, “git-status(1)” for non-porcelain status formats. Defaults to true.

status.displayCommentPrefix

If set to true, Section G.3.138, “git-status(1)” will insert a comment prefix before each output line (starting with *core.commentChar*, i.e. # by default). This was the behavior of Section G.3.138, “git-status(1)” in Git 1.8.4 and previous. Defaults to false.

status.renameLimit

The number of files to consider when performing rename detection in Section G.3.138, “git-status(1)” and Section G.3.28, “git-commit(1)”. Defaults to the value of diff.renameLimit.

status.renames

Whether and how Git detects renames in Section G.3.138, “git-status(1)” and Section G.3.28, “git-commit(1)” . If set to "false", rename detection is disabled. If set to "true", basic rename detection is enabled. If set to "copies" or "copy", Git will detect copies, as well. Defaults to the value of diff.renames.

status.showStash

If set to true, Section G.3.138, “git-status(1)” will display the number of entries currently stashed away. Defaults to false.

status.showUntrackedFiles

By default, Section G.3.138, “git-status(1)” and Section G.3.28, “git-commit(1)” show files which are not currently tracked by Git. Directories which contain only untracked files, are shown with the directory name only. Showing untracked files means that Git needs to lstat() all the files in the whole repository, which might be slow on some systems. So, this variable controls how the commands display the untracked files. Possible values are:

- *no* - Show no untracked files.
- *normal* - Show untracked files and directories.

- *all* - Show also individual files in untracked directories.

If this variable is not specified, it defaults to *normal*. All usual spellings for Boolean value *true* are taken as *normal* and *false* as *no*. This variable can be overridden with the -u|--untracked-files option of Section G.3.138, “git-status(1)” and Section G.3.28, “git-commit(1)”.

status.submoduleSummary

Defaults to false. If this is set to a non-zero number or true (identical to -1 or an unlimited number), the submodule summary will be enabled and a summary of commits for modified submodules will be shown (see --summary-limit option of Section G.3.141, “git-submodule(1)”). Please note that the summary output command will be suppressed for all submodules when *diff.ignoreSubmodules* is set to *all* or only for those submodules where *submodule.<name>.ignore=all*. The only exception to that rule is that status and commit will show staged submodule changes. To also view the summary for ignored submodules you can either use the --ignore-submodules=dirty command-line option or the *git submodule summary* command, which shows a similar output but does not honor these settings.

submodule.<name>.url

The URL for a submodule. This variable is copied from the .gitmodules file to the git config via *git submodule init*. The user can change the configured URL before obtaining the submodule via *git submodule update*. If neither submodule.<name>.active nor submodule.active are set, the presence of this variable is used as a fallback to indicate whether the submodule is of interest to git commands. See Section G.3.141, “git-submodule(1)” and Section G.4.10, “gitmodules(5)” for details.

submodule.<name>.update

The method by which a submodule is updated by *git submodule update*, which is the only affected command, others such as *git checkout --recurse-submodules* are unaffected. It exists for historical reasons, when *git submodule* was the only command to interact with submodules; settings like *submodule.active* and *pull.rebase* are more specific. It is populated by *git submodule init* from the Section G.4.10, “gitmodules(5)” file. See description of *update* command in Section G.3.141, “git-submodule(1)”.

submodule.<name>.branch

The remote branch name for a submodule, used by *git submodule update --remote*. Set this option to override the value found in the *.gitmodules* file. See Section G.3.141, “git-submodule(1)” and Section G.4.10, “gitmodules(5)” for details.

submodule.<name>.fetchRecurseSubmodules

This option can be used to control recursive fetching of this submodule. It can be overridden by using the --[no-]recurse-submodules command-line option to "git fetch" and "git pull". This setting will override that from in the Section G.4.10, “gitmodules(5)” file.

submodule.<name>.ignore

Defines under what circumstances "git status" and the diff family show a submodule as modified. When set to "all", it will never be considered modified (but it will nonetheless show up in the output of status and commit when it has been staged), "dirty" will ignore all changes to the submodule's work tree and takes only differences between the HEAD of the submodule and the commit recorded in the superproject into account. "untracked" will additionally let submodules with modified tracked files in their work tree show up. Using "none" (the default when this option is not set) also shows submodules that have untracked files in their work tree as changed. This setting overrides any setting made in .gitmodules for this submodule, both settings can be overridden on the command line by using the "--ignore-submodules" option. The *git submodule* commands are not affected by this setting.

submodule.<name>.active

Boolean value indicating if the submodule is of interest to git commands. This config option takes precedence over the submodule.active config option. See Section G.4.15, “gitsubmodules(7)” for details.

submodule.active

A repeated field which contains a pathspec used to match against a submodule's path to determine if the submodule is of interest to git commands. See Section G.4.15, “gitsubmodules(7)” for details.

submodule.recurse

A boolean indicating if commands should enable the *--recurse-submodules* option by default. Defaults to false.

When set to true, it can be deactivated via the *--no-recurse-submodules* option. Note that some Git commands lacking this option may call some of the above commands affected by *submodule.recurse*; for instance *git remote update* will call *git fetch* but does not have a *--no-recurse-submodules* option. For these commands a workaround is to temporarily change the configuration value by using *git -c submodule.recurse=0*.

The following list shows the commands that accept *--recurse-submodules* and whether they are supported by this setting.

- *checkout*, *fetch*, *grep*, *pull*, *push*, *read-tree*, *reset*, *restore* and *switch* are always supported.
- *clone* and *ls-files* are not supported.
- *branch* is supported only if *submodule.propagateBranches* is enabled

submodule.propagateBranches

[EXPERIMENTAL] A boolean that enables branching support when using *--recurse-submodules* or *submodule.recurse=true*. Enabling this will allow certain commands to accept *--recurse-submodules* and certain commands that already accept *--recurse-submodules* will now consider branches. Defaults to false.

submodule.fetchJobs

Specifies how many submodules are fetched/cloned at the same time. A positive integer allows up to that number of submodules fetched in parallel. A value of 0 will give some reasonable default. If unset, it defaults to 1.

submodule.alternateLocation

Specifies how the submodules obtain alternates when submodules are cloned. Possible values are *no*, *superproject*. By default *no* is assumed, which doesn't add references. When the value is set to *superproject* the submodule to be cloned computes its alternates location relative to the superprojects alternate.

submodule.alternateErrorStrategy

Specifies how to treat errors with the alternates for a submodule as computed via *submodule.alternateLocation*. Possible values are *ignore*, *info*, *die*. Default is *die*. Note that if set to *ignore* or *info*, and if there is an error with the computed alternate, the clone proceeds as if no alternate was specified.

tag.forceSignAnnotated

A boolean to specify whether annotated tags created should be GPG signed. If *--annotate* is specified on the command line, it takes precedence over this option.

tag.sort

This variable controls the sort ordering of tags when displayed by Section G.3.144, “git-tag(1)”. Without the "--sort=<value>" option provided, the value of this variable will be used as the default.

tag.gpgSign

A boolean to specify whether all tags should be GPG signed. Use of this option when running in an automated script can result in a large number of tags being signed. It is therefore convenient to use an agent to avoid typing your gpg passphrase several times. Note that this option doesn't affect tag signing behavior enabled by "-u <keyid>" or "--local-user=<keyid>" options.

tar.umask

This variable can be used to restrict the permission bits of tar archive entries. The default is 0002, which turns off the world write bit. The special value "user" indicates that the archiving user's umask will be used instead. See umask(2) and Section G.3.7, “git-archive(1)”.

Trace2 config settings are only read from the system and global config files; repository local and worktree config files and *-c* command line arguments are not respected.

trace2.normalTarget

This variable controls the normal target destination. It may be overridden by the *GIT_TRACE2* environment variable. The following table shows possible values.

trace2.perfTarget

This variable controls the performance target destination. It may be overridden by the *GIT_TRACE2_PERF* environment variable. The following table shows possible values.

trace2.eventTarget

This variable controls the event target destination. It may be overridden by the *GIT_TRACE2_EVENT* environment variable. The following table shows possible values.

- *0* or *false* - Disables the target.
- *1* or *true* - Writes to *STDERR*.
- *[2-9]* - Writes to the already opened file descriptor.
- *<absolute-pathname>* - Writes to the file in append mode. If the target already exists and is a directory, the traces will be written to files (one per process) underneath the given directory.
- *af_unix:[<socket-type>:]<absolute-pathname>* - Write to a Unix DomainSocket (on platforms that support them). Socket type can be either *stream* or *dgram*; if omitted Git will try both.

trace2.normalBrief

Boolean. When true *time*, *filename*, and *line* fields are omitted from normal output. May be overridden by the *GIT_TRACE2_BRIEF* environment variable. Defaults to false.

trace2.perfBrief

Boolean. When true *time*, *filename*, and *line* fields are omitted from PERF output. May be overridden by the *GIT_TRACE2_PERF_BRIEF* environment variable. Defaults to false.

trace2.eventBrief

Boolean. When true *time*, *filename*, and *line* fields are omitted from event output. May be overridden by the *GIT_TRACE2_EVENT_BRIEF* environment variable. Defaults to false.

trace2.eventNesting

Integer. Specifies desired depth of nested regions in the event output. Regions deeper than this value will be omitted. May be overridden by the *GIT_TRACE2_EVENT_NESTING* environment variable. Defaults to 2.

trace2.configParams

A comma-separated list of patterns of "important" config settings that should be recorded in the trace2 output. For example, *core.*,remote.*.url* would cause the trace2 output to contain events listing each configured remote. May be overridden by the *GIT_TRACE2_CONFIG_PARAMS* environment variable. Unset by default.

trace2.envVars

A comma-separated list of "important" environment variables that should be recorded in the trace2 output. For example, *GIT_HTTP_USER_AGENT,GIT_CONFIG* would cause the trace2 output to contain events listing the overrides for HTTP user agent and the location of the Git configuration file (assuming any are set). May be overridden by the *GIT_TRACE2_ENV_VARS* environment variable. Unset by default.

trace2.destinationDebug

Boolean. When true Git will print error messages when a trace target destination cannot be opened for writing. By default, these errors are suppressed and tracing is silently disabled. May be overridden by the *GIT_TRACE2_DST_DEBUG* environment variable.

trace2.maxFiles

Integer. When writing trace files to a target directory, do not write additional traces if doing so would exceed this many files. Instead, write a sentinel file that will block further tracing to this directory. Defaults to 0, which disables this check.

transfer.credentialsInUrl

A configured URL can contain plaintext credentials in the form *<protocol>://<user>:<password>@<domain>/<path>*. You may want to warn or forbid the use of such configuration (in favor of using Section G.3.31, “git-credential(1)”). This will be used on Section G.3.24, “git-clone(1)”, Section G.3.49, “git-fetch(1)”, Section G.3.103, “git-push(1)”, and any other direct use of the configured URL.

Note that this is currently limited to detecting credentials in *remote.<name>.url* configuration; it won't detect credentials in *remote.<name>.pushurl* configuration.

You might want to enable this to prevent inadvertent credentials exposure, e.g. because:

- The OS or system where you're running git may not provide a way or otherwise allow you to configure the permissions of the configuration file where the username and/or password are stored.
- Even if it does, having such data stored "at rest" might expose you in other ways, e.g. a backup process might copy the data to another system.
- The git programs will pass the full URL to one another as arguments on the command-line, meaning the credentials will be exposed to other unprivileged users on systems that allow them to see the full process list of other users. On linux the "hidepid" setting documented in procfs(5) allows for configuring this behavior.

  If such concerns don't apply to you then you probably don't need to be concerned about credentials exposure due to storing sensitive data in git's configuration files. If you do want to use this, set *transfer.credentialsInUrl* to one of these values:

- *allow* (default): Git will proceed with its activity without warning.
- *warn*: Git will write a warning message to *stderr* when parsing a URL with a plaintext credential.
- *die*: Git will write a failure message to *stderr* when parsing a URL with a plaintext credential.

transfer.fsckObjects

When *fetch.fsckObjects* or *receive.fsckObjects* are not set, the value of this variable is used instead. Defaults to false.

When set, the fetch or receive will abort in the case of a malformed object or a link to a nonexistent object. In addition, various other issues are checked for, including legacy issues (see *fsck.<msg-id>*), and potential security issues like the existence of a *.GIT* directory or a malicious *.gitmodules* file (see the release notes for v2.2.1 and v2.17.1 for details). Other sanity and security checks may be added in future releases.

On the receiving side, failing fsckObjects will make those objects unreachable, see "QUARANTINE ENVIRONMENT" in Section G.3.108, “git-receive-pack(1)”. On the fetch side, malformed objects will instead be left unreferenced in the repository.

Due to the non-quarantine nature of the *fetch.fsckObjects* implementation it cannot be relied upon to leave the object store clean like *receive.fsckObjects* can.

As objects are unpacked they're written to the object store, so there can be cases where malicious objects get introduced even though the "fetch" failed, only to have a subsequent "fetch" succeed because only new incoming objects are checked, not those that have already been written to the object store. That difference in behavior should not be relied upon. In the future, such objects may be quarantined for "fetch" as well.

For now, the paranoid need to find some way to emulate the quarantine environment if they'd like the same protection as "push". E.g. in the case of an internal mirror do the mirroring in two steps, one to fetch the untrusted objects, and then do a second "push" (which will use the quarantine) to another internal repo, and have internal clients consume this pushed-to repository, or embargo internal fetches and only allow them once a full "fsck" has run (and no new fetches have happened in the meantime).

transfer.hideRefs

String(s) *receive-pack* and *upload-pack* use to decide which refs to omit from their initial advertisements. Use more than one definition to specify multiple prefix strings. A ref that is under the hierarchies listed in the value of this variable is excluded, and is hidden when responding to *git push* or *git fetch*. See *receive.hideRefs* and *uploadpack.hideRefs* for program-specific versions of this config.

You may also include a *!* in front of the ref name to negate the entry, explicitly exposing it, even if an earlier entry marked it as hidden. If you have multiple hideRefs values, later entries override earlier ones (and entries in more-specific config files override less-specific ones).

If a namespace is in use, the namespace prefix is stripped from each reference before it is matched against *transfer.hiderefs* patterns. In order to match refs before stripping, add a ^ in front of the ref name. If you combine *!* and ^, *!* must be specified first.

For example, if *refs/heads/master* is specified in *transfer.hideRefs* and the current namespace is *foo*, then *refs/namespaces/foo/refs/heads/master* is omitted from the advertisements. If *uploadpack.allowRefInWant* is set, *upload-pack* will treat *want-ref refs/heads/master* in a protocol v2 *fetch* command as if *refs/namespaces/foo/refs/heads/master* did not exist. *receive-pack*, on the other hand, will still advertise the object id the ref is pointing to without mentioning its name (a so-called ".have" line).

Even if you hide refs, a client may still be able to steal the target objects via the techniques described in the "SECURITY" section of the Section G.4.11, “gitnamespaces(7)” man page; it's best to keep private data in a separate repository.

transfer.unpackLimit

When *fetch.unpackLimit* or *receive.unpackLimit* are not set, the value of this variable is used instead. The default value is 100.

transfer.advertiseSID

Boolean. When true, client and server processes will advertise their unique session IDs to their remote counterpart. Defaults to false.

transfer.bundleURI

When *true*, local *git clone* commands will request bundle information from the remote server (if advertised) and download bundles before continuing the clone through the Git protocol. Defaults to *false*.

transfer.advertiseObjectInfo

When *true*, the *object-info* capability is advertised by servers. Defaults to false.

uploadarchive.allowUnreachable

If true, allow clients to use *git archive --remote* to request any tree, whether reachable from the ref tips or not. See the discussion in the "SECURITY" section of Section G.3.150, “git-upload-archive(1)” for more details. Defaults to *false*.

uploadpack.hideRefs

This variable is the same as *transfer.hideRefs*, but applies only to *upload-pack* (and so affects only fetches, not pushes). An attempt to fetch a hidden ref by *git fetch* will fail. See also *uploadpack.allowTipSHA1InWant*.

uploadpack.allowTipSHA1InWant

When *uploadpack.hideRefs* is in effect, allow *upload-pack* to accept a fetch request that asks for an object at the tip of a hidden ref (by default, such a request is rejected). See also *uploadpack.hideRefs*. Even if this is false, a client may be able to steal objects via the techniques described in the "SECURITY" section of the Section G.4.11, “gitnamespaces(7)” man page; it's best to keep private data in a separate repository.

uploadpack.allowReachableSHA1InWant

Allow *upload-pack* to accept a fetch request that asks for an object that is reachable from any ref tip. However, note that calculating object reachability is computationally expensive. Defaults to *false*. Even if this is false, a client may be able to steal objects via the techniques described in the "SECURITY" section of the Section G.4.11, “gitnamespaces(7)” man page; it's best to keep private data in a separate repository.

uploadpack.allowAnySHA1InWant

Allow *upload-pack* to accept a fetch request that asks for any object at all. Defaults to *false*.

uploadpack.keepAlive

When *upload-pack* has started *pack-objects*, there may be a quiet period while *pack-objects* prepares the pack. Normally it would output progress information, but if *--quiet* was used for the fetch, *pack-objects* will output nothing at all until the pack data begins. Some clients and networks may consider the server to be hung and give up. Setting this option instructs *upload-pack* to send an empty keepalive packet every *uploadpack.keepAlive* seconds. Setting this option to 0 disables keepalive packets entirely. The default is 5 seconds.

uploadpack.packObjectsHook

If this option is set, when *upload-pack* would run *git pack-objects* to create a packfile for a client, it will run this shell command instead. The *pack-objects* command and arguments it *would* have run (including the *git pack-objects* at the beginning) are appended to the shell command. The stdin and stdout of the hook are treated as if *pack-objects* itself was run. I.e., *upload-pack* will feed input intended for *pack-objects* to the hook, and expects a completed packfile on stdout.

Note that this configuration variable is only respected when it is specified in protected configuration (see the section called “SCOPES”). This is a safety measure against fetching from untrusted repositories.

uploadpack.allowFilter

If this option is set, *upload-pack* will support partial clone and partial fetch object filtering.

uploadpackfilter.allow

Provides a default value for unspecified object filters (see: the below configuration variable). If set to *true*, this will also enable all filters which get added in the future. Defaults to *true*.

uploadpackfilter.<filter>.allow

Explicitly allow or ban the object filter corresponding to *<filter>*, where *<filter>* may be one of: *blob:none*, *blob:limit*, *object:type*, *tree*, *sparse:oid*, or *combine*. If using combined filters, both *combine* and all of the nested filter kinds must be allowed. Defaults to *uploadpackfilter.allow*.

uploadpackfilter.tree.maxDepth

Only allow *--filter=tree:<n>* when *<n>* is no more than the value of *uploadpackfilter.tree.maxDepth*. If set, this also implies *uploadpackfilter.tree.allow=true*, unless this configuration variable had already been set. Has no effect if unset.

uploadpack.allowRefInWant

If this option is set, *upload-pack* will support the *ref-in-want* feature of the protocol version 2 *fetch* command. This feature is intended for the benefit of load-balanced servers which may not have the same view of what OIDs their refs point to due to replication delay.

url.<base>.insteadOf

Any URL that starts with this value will be rewritten to start, instead, with <base>. In cases where some site serves a large number of repositories, and serves them with multiple access methods, and some users need to use different access methods, this feature allows people to specify any of the equivalent URLs and have Git automatically rewrite the URL to the best alternative for the particular user, even for a never-before-seen repository on the site. When more than one insteadOf strings match a given URL, the longest match is used.

Note that any protocol restrictions will be applied to the rewritten URL. If the rewrite changes the URL to use a custom protocol or remote helper, you may need to adjust the *protocol.*.allow* config to permit the request. In particular, protocols you expect to use for submodules must be set to *always* rather than the default of *user*. See the description of *protocol.allow* above.

url.<base>.pushInsteadOf

Any URL that starts with this value will not be pushed to; instead, it will be rewritten to start with <base>, and the resulting URL will be pushed to. In cases where some site serves a large number of repositories, and serves them with multiple access methods, some of which do not allow push, this feature allows people to specify a pull-only URL and have Git automatically use an appropriate URL to push, even for a never-before-seen repository on the site. When more than one pushInsteadOf strings match a given URL, the longest match is used. If a remote has an explicit pushurl, Git will ignore this setting for that remote.

user.name , user.email , author.name , author.email , committer.name , committer.email

The *user.name* and *user.email* variables determine what ends up in the *author* and *committer* fields of commit objects. If you need the *author* or *committer* to be different, the *author.name*, *author.email*, *committer.name*, or *committer.email* variables can be set. All of these can be overridden by the *GIT_AUTHOR_NAME*, *GIT_AUTHOR_EMAIL*, *GIT_COMMITTER_NAME*, *GIT_COMMITTER_EMAIL*, and *EMAIL* environment variables.

Note that the *name* forms of these variables conventionally refer to some form of a personal name. See Section G.3.28, “git-commit(1)” and the environment variables section of Section G.3.1, “git(1)” for more information on these settings and the *credential.username* option if you're looking for authentication credentials instead.

user.useConfigOnly

Instruct Git to avoid trying to guess defaults for *user.email* and *user.name*, and instead retrieve the values only from the configuration. For example, if you have multiple email addresses and would like to use a different one for each repository, then with this configuration option set to *true* in the global config along with a name, Git will prompt you to set up an email before making new commits in a newly cloned repository. Defaults to *false*.

user.signingKey

If Section G.3.144, “git-tag(1)” or Section G.3.28, “git-commit(1)” is not selecting the key you want it to automatically when creating a signed tag or commit, you can override the default selection with this variable. This option is passed unchanged to gpg's --local-user parameter, so you may specify a key using any method that gpg supports. If gpg.format is set to *ssh* this can contain the path to either your private ssh key or the public key when ssh-agent is used. Alternatively it can contain a public key prefixed with *key::* directly (e.g.:

"key::ssh-rsa XXXXXX identifier"). The private key needs to be available via ssh-agent. If not set Git will call gpg.ssh.defaultKeyCommand (e.g.: "ssh-add -L") and try to use the first key available. For backward compatibility, a raw key which begins with "ssh-", such as "ssh-rsa XXXXXX identifier", is treated as "key::ssh-rsa XXXXXX identifier", but this form is deprecated; use the *key::* form instead.

versionsort.prereleaseSuffix (deprecated)

Deprecated alias for *versionsort.suffix*. Ignored if *versionsort.suffix* is set.

versionsort.suffix

Even when version sort is used in Section G.3.144, “git-tag(1)”, tagnames with the same base version but different suffixes are still sorted lexicographically, resulting e.g. in prerelease tags appearing after the main release (e.g. "1.0-rc1" after "1.0"). This variable can be specified to determine the sorting order of tags with different suffixes.

By specifying a single suffix in this variable, any tagname containing that suffix will appear before the corresponding main release. E.g. if the variable is set to "-rc", then all "1.0-rcX" tags will appear before "1.0". If specified multiple times, once per suffix, then the order of suffixes in the configuration will determine the sorting order of tagnames with those suffixes. E.g. if "-pre" appears before "-rc" in the configuration, then all "1.0-preX" tags will be listed before any "1.0-rcX" tags. The placement of the main release tag relative to tags with various suffixes can be determined by specifying the empty suffix among those other suffixes. E.g. if the suffixes "-rc", "", "-ck", and "-bfs" appear in the configuration in this order, then all "v4.8-rcX" tags are listed first, followed by "v4.8", then "v4.8-ckX" and finally "v4.8-bfsX".

If more than one suffix matches the same tagname, then that tagname will be sorted according to the suffix which starts at the earliest position in the tagname. If more than one different matching suffix starts at that earliest position, then that tagname will be sorted according to the longest of those suffixes. The sorting order between different suffixes is undefined if they are in multiple config files.

web.browser

Specify a web browser that may be used by some commands. Currently only Section G.3.72, “git-instaweb(1)” and Section G.3.63, “git-help(1)” may use it.

worktree.guessRemote

If no branch is specified and neither *-b* nor *-B* nor *--detach* is used, then *git worktree add* defaults to creating a new branch from HEAD. If *worktree.guessRemote* is set to true, *worktree add* tries to find a remote-tracking branch whose name uniquely matches the new branch name. If such a branch exists, it is checked out and set as "upstream" for the new branch. If no such match can be found, it falls back to creating a new branch from the current HEAD.

BUGS

When using the deprecated *[section.subsection]* syntax, changing a value will result in adding a multi-line key instead of a change, if the subsection is given with at least one uppercase character. For example when the config looks like

```
[section.subsection]
  key = value1
```

and running *git config section.Subsection.key value2* will result in

```
[section.subsection]
  key = value1
  key = value2
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.30. git-count-objects(1)

2024-07-02
Revision History

2024-07-02

NAME

git-count-objects - Count unpacked number of objects and their disk consumption

SYNOPSIS

git count-objects [-v] [-H | --human-readable]

DESCRIPTION

Counts the number of unpacked object files and disk space consumed by them, to help you decide when it is a good time to repack.

OPTIONS

-v , --verbose

Provide more detailed reports:

count: the number of loose objects

size: disk space consumed by loose objects, in KiB (unless -H is specified)

in-pack: the number of in-pack objects

size-pack: disk space consumed by the packs, in KiB (unless -H is specified)

prune-packable: the number of loose objects that are also present in the packs. These objects could be pruned using *git prune-packed*.

garbage: the number of files in the object database that are neither valid loose objects nor valid packs

size-garbage: disk space consumed by garbage files, in KiB (unless -H is specified)

alternate: absolute path of alternate object databases; may appear multiple times, one line per path. Note that if the path contains non-printable characters, it may be surrounded by double-quotes and contain C-style backslashed escape sequences.

-H , --human-readable

Print sizes in human readable format

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.31. git-credential(1)

2024-07-02
Revision History

2024-07-02

NAME

git-credential - Retrieve and store user credentials

SYNOPSIS

```
'git credential' (fill|approve|reject)
```

DESCRIPTION

Git has an internal interface for storing and retrieving credentials from system-specific helpers, as well as prompting the user for usernames and passwords. The git-credential command exposes this interface to scripts which may want to retrieve, store, or prompt for credentials in the same manner as Git. The design of this scriptable interface models the internal C API; see credential.h for more background on the concepts.

git-credential takes an "action" option on the command-line (one of *fill*, *approve*, or *reject*) and reads a credential description on stdin (see INPUT/OUTPUT FORMAT).

If the action is *fill*, git-credential will attempt to add "username" and "password" attributes to the description by reading config files, by contacting any configured credential helpers, or by prompting the user. The username and password attributes of the credential description are then printed to stdout together with the attributes already provided.

If the action is *approve*, git-credential will send the description to any configured credential helpers, which may store the credential for later use.

If the action is *reject*, git-credential will send the description to any configured credential helpers, which may erase any stored credentials matching the description.

If the action is *approve* or *reject*, no output should be emitted.

TYPICAL USE OF GIT CREDENTIAL

An application using git-credential will typically use *git credential* following these steps:

1. Generate a credential description based on the context.

   For example, if we want a password for *https://example.com/foo.git*, we might generate the following credential description (don't forget the blank line at the end; it tells *git credential* that the application finished feeding all the information it has):

   ```
   protocol=https
   host=example.com
   path=foo.git
   ```

2. Ask git-credential to give us a username and password for this description. This is done by running *git credential fill*, feeding the description from step (1) to its standard input. The complete credential description (including the credential per se, i.e. the login and password) will be produced on standard output, like:

   ```
   protocol=https
   host=example.com
   username=bob
   password=secr3t
   ```

   In most cases, this means the attributes given in the input will be repeated in the output, but Git may also modify the credential description, for example by removing the *path* attribute when the protocol is HTTP(s) and *credential.useHttpPath* is false.

   If the *git credential* knew about the password, this step may not have involved the user actually typing this password (the user may have typed a password to unlock the keychain instead, or no user interaction was done if the keychain was already unlocked) before it returned *password=secr3t*.

3. Use the credential (e.g., access the URL with the username and password from step (2)), and see if it's accepted.

4. Report on the success or failure of the password. If the credential allowed the operation to complete successfully, then it can be marked with an "approve" action to tell *git credential* to reuse it in its next invocation. If the credential was rejected during the operation, use the "reject" action so that *git credential* will ask for a new password in its next invocation. In either case, *git credential* should be fed with the credential description obtained from step (2) (which also contains the fields provided in step (1)).

INPUT/OUTPUT FORMAT

git credential reads and/or writes (depending on the action used) credential information in its standard input/output. This information can correspond either to keys for which *git credential* will obtain the login information (e.g. host, protocol, path), or to the actual credential data to be obtained (username/password).

The credential is split into a set of named attributes, with one attribute per line. Each attribute is specified by a key-value pair, separated by an = (equals) sign, followed by a newline.

The key may contain any bytes except =, newline, or NUL. The value may contain any bytes except newline or NUL.

Attributes with keys that end with C-style array brackets *[]* can have multiple values. Each instance of a multi-valued attribute forms an ordered list of values - the order of the repeated attributes defines the order of the values. An empty multi-valued attribute (*key[]=\n*) acts to clear any previous entries and reset the list.

In all cases, all bytes are treated as-is (i.e., there is no quoting, and one cannot transmit a value with newline or NUL in it). The list of attributes is terminated by a blank line or end-of-file.

Git understands the following attributes:

protocol

The protocol over which the credential will be used (e.g., *https*).

host

The remote hostname for a network credential. This includes the port number if one was specified (e.g., "example.com:8088").

path

The path with which the credential will be used. E.g., for accessing a remote https repository, this will be the repository's path on the server.

username

The credential's username, if we already have one (e.g., from a URL, the configuration, the user, or from a previously run helper).

password

The credential's password, if we are asking it to be stored.

password_expiry_utc

Generated passwords such as an OAuth access token may have an expiry date. When reading credentials from helpers, *git credential fill* ignores expired passwords. Represented as Unix time UTC, seconds since 1970.

oauth_refresh_token

An OAuth refresh token may accompany a password that is an OAuth access token. Helpers must treat this attribute as confidential like the password attribute. Git itself has no special behaviour for this attribute.

url

When this special attribute is read by *git credential*, the value is parsed as a URL and treated as if its constituent parts were read (e.g., *url=https://example.com* would behave as if *protocol=https* and *host=example.com* had been provided). This can help callers avoid parsing URLs themselves.

Note that specifying a protocol is mandatory and if the URL doesn't specify a hostname (e.g., "cert:///path/to/file") the credential will contain a hostname attribute whose value is an empty string.

Components which are missing from the URL (e.g., there is no username in the example above) will be left unset.

wwwauth[]

When an HTTP response is received by Git that includes one or more *WWW-Authenticate* authentication headers, these will be passed by Git to credential helpers.

Each *WWW-Authenticate* header value is passed as a multi-valued attribute *wwwauth[]*, where the order of the attributes is the same as they appear in the HTTP response. This attribute is *one-way* from Git to pass additional information to credential helpers.

Unrecognised attributes are silently discarded.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.32. git-credential-cache--daemon(1)

2024-07-02
Revision History
2024-07-02

NAME

git-credential-cache--daemon - Temporarily store user credentials in memory

SYNOPSIS

git credential-cache--daemon [--debug] <socket-path>

DESCRIPTION

> **Note**
>
> You probably don't want to invoke this command yourself; it is started automatically when you use Section G.3.33, “git-credential-cache(1)”.

This command listens on the Unix domain socket specified by *<socket-path>* for *git-credential-cache* clients. Clients may store and retrieve credentials. Each credential is held for a timeout specified by the client; once no credentials are held, the daemon exits.

If the *--debug* option is specified, the daemon does not close its stderr stream, and may output extra diagnostics to it even after it has begun listening for clients.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.33. git-credential-cache(1)

2024-07-02
Revision History
2024-07-02

NAME

git-credential-cache - Helper to temporarily store passwords in memory

SYNOPSIS

```
git config credential.helper 'cache [<options>]'
```

DESCRIPTION

This command caches credentials for use by future Git programs. The stored credentials are kept in memory of the cache-daemon process (instead of being written to a file) and are forgotten after a configurable timeout. Credentials are forgotten sooner if the cache-daemon dies, for example if the system restarts. The cache is accessible over a Unix domain socket, restricted to the current user by filesystem permissions.

You probably don't want to invoke this command directly; it is meant to be used as a credential helper by other parts of Git. See Section G.4.3, “gitcredentials(7)” or *EXAMPLES* below.

OPTIONS

--timeout <seconds>

Number of seconds to cache credentials (default: 900).

--socket <path>

Use *<path>* to contact a running cache daemon (or start a new cache daemon if one is not started). Defaults to *$XDG_CACHE_HOME/git/credential/socket* unless *~/.git-credential-cache/* exists in which case *~/.git-credential-cache/socket* is used instead. If your home directory is on a network-mounted filesystem, you may need to change this to a local filesystem. You must specify an absolute path.

CONTROLLING THE DAEMON

If you would like the daemon to exit early, forgetting all cached credentials before their timeout, you can issue an *exit* action:

```
git credential-cache exit
```

EXAMPLES

The point of this helper is to reduce the number of times you must type your username or password. For example:

```
$ git config credential.helper cache
$ git push http://example.com/repo.git
Username: <type your username>
Password: <type your password>

[work for 5 more minutes]
$ git push http://example.com/repo.git
[your credentials are used automatically]
```

You can provide options via the credential.helper configuration variable (this example increases the cache time to 1 hour):

```
$ git config credential.helper 'cache --timeout=3600'
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.34. git-credential-store(1)

2024-07-02

Revision History

2024-07-02

NAME

git-credential-store - Helper to store credentials on disk

SYNOPSIS

```
git config credential.helper 'store [<options>]'
```

DESCRIPTION

> **Note**
>
> Using this helper will store your passwords unencrypted on disk, protected only by filesystem permissions. If this is not an acceptable security tradeoff, try Section G.3.33, “git-credential-cache(1)”, or find a helper that integrates with secure storage provided by your operating system.

This command stores credentials indefinitely on disk for use by future Git programs.

You probably don't want to invoke this command directly; it is meant to be used as a credential helper by other parts of git. See Section G.4.3, “gitcredentials(7)” or *EXAMPLES* below.

OPTIONS

--file=<path>

Use <*path*> to lookup and store credentials. The file will have its filesystem permissions set to prevent other users on the system from reading it, but it will not be encrypted or otherwise protected. If not specified, credentials will be searched for from *~/.git-credentials* and *$XDG_CONFIG_HOME/git/credentials*, and credentials will be written to *~/.git-credentials* if it exists, or *$XDG_CONFIG_HOME/git/credentials* if it exists and the former does not. See also the section called “FILES”.

FILES

If not set explicitly with *--file*, there are two files where git-credential-store will search for credentials in order of precedence:

~/.git-credentials

User-specific credentials file.

$XDG_CONFIG_HOME/git/credentials

Second user-specific credentials file. If *$XDG_CONFIG_HOME* is not set or empty, *$HOME/.config/git/credentials* will be used. Any credentials stored in this file will not be used if *~/.git-credentials* has a matching credential as well. It is a good idea not to create this file if you sometimes use older versions of Git that do not support it.

For credential lookups, the files are read in the order given above, with the first matching credential found taking precedence over credentials found in files further down the list.

Credential storage will by default write to the first existing file in the list. If none of these files exist, *~/.git-credentials* will be created and written to.

When erasing credentials, matching credentials will be erased from all files.

EXAMPLES

The point of this helper is to reduce the number of times you must type your username or password. For example:

```
$ git config credential.helper store
$ git push http://example.com/repo.git
Username: <type your username>
Password: <type your password>
```

```
[several days later]
$ git push http://example.com/repo.git
[your credentials are used automatically]
```

STORAGE FORMAT

The *.git-credentials* file is stored in plaintext. Each credential is stored on its own line as a URL like:

```
https://user:pass@example.com
```

No other kinds of lines (e.g. empty lines or comment lines) are allowed in the file, even though some may be silently ignored. Do not view or edit the file with editors.

When Git needs authentication for a particular URL context, credential-store will consider that context a pattern to match against each entry in the credentials file. If the protocol, hostname, and username (if we already have one) match, then the password is returned to Git. See the discussion of configuration in Section G.4.3, “gitcredentials(7)” for more information.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.35. git-cvsexportcommit(1)

2024-07-02
Revision History
2024-07-02

NAME

git-cvsexportcommit - Export a single commit to a CVS checkout

SYNOPSIS

git cvsexportcommit [-h] [-u] [-v] [-c] [-P] [-p] [-a] [-d <cvsroot>]
[-w <cvs-workdir>] [-W] [-f] [-m <msgprefix>] [<parent-commit>] <commit-id>

DESCRIPTION

Exports a commit from Git to a CVS checkout, making it easier to merge patches from a Git repository into a CVS repository.

Specify the name of a CVS checkout using the -w switch or execute it from the root of the CVS working copy. In the latter case GIT_DIR must be defined. See examples below.

It does its best to do the safe thing, it will check that the files are unchanged and up to date in the CVS checkout, and it will not autocommit by default.

Supports file additions, removals, and commits that affect binary files.

If the commit is a merge commit, you must tell *git cvsexportcommit* what parent the changeset should be done against.

OPTIONS

-c

Commit automatically if the patch applied cleanly. It will not commit if any hunks fail to apply or there were other problems.

-p

Be pedantic (paranoid) when applying patches. Invokes patch with --fuzz=0

-a

Add authorship information. Adds Author line, and Committer (if different from Author) to the message.

-d

Set an alternative CVSROOT to use. This corresponds to the CVS -d parameter. Usually users will not want to set this, except if using CVS in an asymmetric fashion.

-f

Force the merge even if the files are not up to date.

-P

Force the parent commit, even if it is not a direct parent.

-m

Prepend the commit message with the provided prefix. Useful for patch series and the like.

-u

Update affected files from CVS repository before attempting export.

-k

Reverse CVS keyword expansion (e.g. $Revision: 1.2.3.4$ becomes $Revision$) in working CVS checkout before applying patch.

-w

Specify the location of the CVS checkout to use for the export. This option does not require GIT_DIR to be set before execution if the current directory is within a Git repository. The default is the value of *cvsexport-commit.cvsdir*.

-W

Tell cvsexportcommit that the current working directory is not only a Git checkout, but also the CVS checkout. Therefore, Git will reset the working directory to the parent commit before proceeding.

-v

Verbose.

CONFIGURATION

cvsexportcommit.cvsdir

The default location of the CVS checkout to use for the export.

EXAMPLES

Merge one patch into CVS

```
$ export GIT_DIR=~/project/.git
$ cd ~/project_cvs_checkout
$ git cvsexportcommit -v <commit-sha1>
$ cvs commit -F .msg <files>
```

Merge one patch into CVS (-c and -w options). The working directory is within the Git Repo

```
$ git cvsexportcommit -v -c -w ~/project_cvs_checkout <commit-sha1>
```

Merge pending patches into CVS automatically -- only if you really know what you are doing

```
$ export GIT_DIR=~/project/.git
$ cd ~/project_cvs_checkout
$ git cherry cvshead myhead | sed -n 's/^+ //p' | xargs -l1 git
 cvsexportcommit -c -p -v
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.36. git-cvsimport(1)

2024-07-02
Revision History
2024-07-02

NAME

git-cvsimport - Salvage your data out of another SCM people love to hate

SYNOPSIS

git cvsimport [-o <branch-for-HEAD>] [-h] [-v] [-d <CVSROOT>]
[-A <author-conv-file>] [-p <options-for-cvsps>] [-P <file>]
[-C <git-repository>] [-z <fuzz>] [-i] [-k] [-u] [-s <subst>]
[-a] [-m] [-M <regex>] [-S <regex>] [-L <commit-limit>]
[-r <remote>] [-R] [<CVS-module>]

DESCRIPTION

WARNING: *git cvsimport* uses cvsps version 2, which is considered deprecated; it does not work with cvsps version 3 and later. If you are performing a one-shot import of a CVS repository consider using cvs2git [http://cvs2svn.tigris.org/cvs2git.html] or cvs-fast-export [https://gitlab.com/esr/cvs-fast-export].

Imports a CVS repository into Git. It will either create a new repository, or incrementally import into an existing one.

Splitting the CVS log into patch sets is done by *cvsps*. At least version 2.1 is required.

WARNING: for certain situations the import leads to incorrect results. Please see the section ISSUES for further reference.

You should **never** do any work of your own on the branches that are created by *git cvsimport*. By default initial import will create and populate a "master" branch from the CVS repository's main branch which you're free to work with; after that, you need to *git merge* incremental imports, or any CVS branches, yourself. It is advisable to specify a named remote via -r to separate and protect the incoming branches.

If you intend to set up a shared public repository that all developers can read/write, or if you want to use Section G.3.37, “git-cvsserver(1)”, then you probably want to make a bare clone of the imported repository, and use the clone as the shared repository. See Section G.2.4, “gitcvs-migration(7)”.

OPTIONS

-v

Verbosity: let *cvsimport* report what it is doing.

-d <CVSROOT>

The root of the CVS archive. May be local (a simple path) or remote; currently, only the :local:, :ext: and :pserver: access methods are supported. If not given, *git cvsimport* will try to read it from *CVS/Root*. If no such file exists, it checks for the *CVSROOT* environment variable.

<CVS-module>

The CVS module you want to import. Relative to <CVSROOT>. If not given, *git cvsimport* tries to read it from *CVS/Repository*.

-C <target-dir>

The Git repository to import to. If the directory doesn't exist, it will be created. Default is the current directory.

-r <remote>

The Git remote to import this CVS repository into. Moves all CVS branches into remotes/<remote>/<branch> akin to the way *git clone* uses *origin* by default.

-o <branch-for-HEAD>

When no remote is specified (via -r) the *HEAD* branch from CVS is imported to the *origin* branch within the Git repository, as *HEAD* already has a special meaning for Git. When a remote is specified the *HEAD* branch is named remotes/<remote>/master mirroring *git clone* behaviour. Use this option if you want to import into a different branch.

Use *-o master* for continuing an import that was initially done by the old cvs2git tool.

-i

Import-only: don't perform a checkout after importing. This option ensures the working directory and index remain untouched and will not create them if they do not exist.

-k

Kill keywords: will extract files with *-kk* from the CVS archive to avoid noisy changesets. Highly recommended, but off by default to preserve compatibility with early imported trees.

-u

Convert underscores in tag and branch names to dots.

-s <subst>

Substitute the character "/" in branch names with <subst>

-p <options-for-cvsps>

Additional options for cvsps. The options *-u* and *-A* are implicit and should not be used here.

If you need to pass multiple options, separate them with a comma.

-z <fuzz>

Pass the timestamp fuzz factor to cvsps, in seconds. If unset, cvsps defaults to 300s.

-P <cvsps-output-file>

Instead of calling cvsps, read the provided cvsps output file. Useful for debugging or when cvsps is being handled outside cvsimport.

-m

Attempt to detect merges based on the commit message. This option will enable default regexes that try to capture the source branch name from the commit message.

-M <regex>

Attempt to detect merges based on the commit message with a custom regex. It can be used with *-m* to enable the default regexes as well. You must escape forward slashes.

The regex must capture the source branch name in $1.

This option can be used several times to provide several detection regexes.

-S <regex>

Skip paths matching the regex.

-a

Import all commits, including recent ones. cvsimport by default skips commits that have a timestamp less than 10 minutes ago.

-L <limit>

Limit the number of commits imported. Workaround for cases where cvsimport leaks memory.

-A <author-conv-file>

CVS by default uses the Unix username when writing its commit logs. Using this option and an author-conv-file maps the name recorded in CVS to author name, e-mail and optional time zone:

```
exon=Andreas Ericsson <ae@op5.se>
spawn=Simon Pawn <spawn@frog-pond.org> America/Chicago
```

git cvsimport will make it appear as those authors had their GIT_AUTHOR_NAME and GIT_AUTHOR_E-MAIL set properly all along. If a time zone is specified, GIT_AUTHOR_DATE will have the corresponding offset applied.

For convenience, this data is saved to *$GIT_DIR/cvs-authors* each time the *-A* option is provided and read from that same file each time *git cvsimport* is run.

It is not recommended to use this feature if you intend to export changes back to CVS again later with *git cvsexportcommit*.

-R

Generate a *$GIT_DIR/cvs-revisions* file containing a mapping from CVS revision numbers to newly-created Git commit IDs. The generated file will contain one line for each (filename, revision) pair imported; each line will look like

```
src/widget.c 1.1 1d862f173cdc7325b6fa6d2ae1cfd61fd1b512b7
```

The revision data is appended to the file if it already exists, for use when doing incremental imports.

This option may be useful if you have CVS revision numbers stored in commit messages, bug-tracking systems, email archives, and the like.

-h

Print a short usage message and exit.

OUTPUT

If *-v* is specified, the script reports what it is doing.

Otherwise, success is indicated the Unix way, i.e. by simply exiting with a zero exit status.

ISSUES

Problems related to timestamps:

- If timestamps of commits in the CVS repository are not stable enough to be used for ordering commits changes may show up in the wrong order.
- If any files were ever "cvs import"ed more than once (e.g., import of more than one vendor release) the HEAD contains the wrong content.
- If the timestamp order of different files cross the revision order within the commit matching time window the order of commits may be wrong.

Problems related to branches:

- Branches on which no commits have been made are not imported.
- All files from the branching point are added to a branch even if never added in CVS.
- This applies to files added to the source branch **after** a daughter branch was created: if previously no commit was made on the daughter branch they will erroneously be added to the daughter branch in git.

Problems related to tags:

- Multiple tags on the same revision are not imported.

If you suspect that any of these issues may apply to the repository you want to import, consider using cvs2git:

- cvs2git (part of cvs2svn), *https://subversion.apache.org/*

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.37. git-cvsserver(1)

2024-07-02
Revision History
2024-07-02

NAME

git-cvsserver - A CVS server emulator for Git

SYNOPSIS

SSH:

```
export CVS_SERVER="git cvsserver"
cvs -d :ext:user@server/path/repo.git co <HEAD_name>
```

pserver (/etc/inetd.conf):

```
cvspserver stream tcp nowait nobody /usr/bin/git-cvsserver git-cvsserver pserver
```

Usage:

git-cvsserver [<options>] [pserver|server] [<directory> …]

DESCRIPTION

This application is a CVS emulation layer for Git.

It is highly functional. However, not all methods are implemented, and for those methods that are implemented, not all switches are implemented.

Testing has been done using both the CLI CVS client, and the Eclipse CVS plugin. Most functionality works fine with both of these clients.

OPTIONS

All these options obviously only make sense if enforced by the server side. They have been implemented to resemble the Section G.3.38, “git-daemon(1)” options as closely as possible.

--base-path <path>

Prepend *path* to requested CVSROOT

--strict-paths

Don't allow recursing into subdirectories

--export-all

Don't check for *gitcvs.enabled* in config. You also have to specify a list of allowed directories (see below) if you want to use this option.

-V , --version

Print version information and exit

-h , -H , --help

Print usage information and exit

<directory>

The remaining arguments provide a list of directories. If no directories are given, then all are allowed. Repositories within these directories still require the *gitcvs.enabled* config option, unless *--export-all* is specified.

LIMITATIONS

CVS clients cannot tag, branch or perform Git merges.

git-cvsserver maps Git branches to CVS modules. This is very different from what most CVS users would expect since in CVS modules usually represent one or more directories.

INSTALLATION

1. If you are going to offer CVS access via pserver, add a line in /etc/inetd.conf like

   ```
   cvspserver stream tcp nowait nobody git-cvsserver pserver
   ```

   Note: Some inetd servers let you specify the name of the executable independently of the value of argv[0] (i.e. the name the program assumes it was executed with). In this case the correct line in /etc/inetd.conf looks like

   ```
   cvspserver stream tcp nowait nobody /usr/bin/git-cvsserver git-
   cvsserver pserver
   ```

   Only anonymous access is provided by pserver by default. To commit you will have to create pserver accounts, simply add a gitcvs.authdb setting in the config file of the repositories you want the cvsserver to allow writes to, for example:

   ```
   [gitcvs]
        authdb = /etc/cvsserver/passwd
   ```

   The format of these files is username followed by the encrypted password, for example:

```
myuser:sqkNi8zPf01HI
myuser:$1$9K7FzU28$VfF6EoPYCJEYcVQwATgOP/
myuser:$5$.NqmNH1vwfzGpV8B$znZIcumu1tNLATgV2l6e1/mY8RzhUDHMOaVOeL1cxV3
```

You can use the *htpasswd* facility that comes with Apache to make these files, but only with the -d option (or -B if your system supports it).

Preferably use the system specific utility that manages password hash creation in your platform (e.g. mkpasswd in Linux, encrypt in OpenBSD or pwhash in NetBSD) and paste it in the right location.

Then provide your password via the pserver method, for example:

```
cvs -d:pserver:someuser:somepassword@server:/path/repo.git co
<HEAD_name>
```

No special setup is needed for SSH access, other than having Git tools in the PATH. If you have clients that do not accept the CVS_SERVER environment variable, you can rename *git-cvsserver* to *cvs*.

Note: Newer CVS versions (>= 1.12.11) also support specifying CVS_SERVER directly in CVSROOT like

```
cvs -d ":ext;CVS_SERVER=git cvsserver:user@server/path/repo.git" co
<HEAD_name>
```

This has the advantage that it will be saved in your *CVS/Root* files and you don't need to worry about always setting the correct environment variable. SSH users restricted to *git-shell* don't need to override the default with CVS_SERVER (and shouldn't) as *git-shell* understands *cvs* to mean *git-cvsserver* and pretends that the other end runs the real *cvs* better.

2. For each repo that you want accessible from CVS you need to edit config in the repo and add the following section.

```
[gitcvs]
     enabled=1
     # optional for debugging
     logFile=/path/to/logfile
```

Note: you need to ensure each user that is going to invoke *git-cvsserver* has write access to the log file and to the database (see Database Backend. If you want to offer write access over SSH, the users of course also need write access to the Git repository itself.

You also need to ensure that each repository is "bare" (without a Git index file) for *cvs commit* to work. See Section G.2.4, “gitcvs-migration(7)”.

All configuration variables can also be overridden for a specific method of access. Valid method names are "ext" (for SSH access) and "pserver". The following example configuration would disable pserver access while still allowing access over SSH.

```
[gitcvs]
     enabled=0

[gitcvs "ext"]
     enabled=1
```

3. If you didn't specify the CVSROOT/CVS_SERVER directly in the checkout command, automatically saving it in your *CVS/Root* files, then you need to set them explicitly in your environment. CVSROOT should be set as per normal, but the directory should point at the appropriate Git repo. As above, for SSH clients *not* restricted to *git-shell*, CVS_SERVER should be set to *git-cvsserver*.

```
export CVSROOT=:ext:user@server:/var/git/project.git
export CVS_SERVER="git cvsserver"
```

4. For SSH clients that will make commits, make sure their server-side .ssh/environment files (or .bashrc, etc., according to their specific shell) export appropriate values for GIT_AUTHOR_NAME, GIT_AUTHOR_E-MAIL, GIT_COMMITTER_NAME, and GIT_COMMITTER_EMAIL. For SSH clients whose login shell is bash, .bashrc may be a reasonable alternative.

5. Clients should now be able to check out the project. Use the CVS *module* name to indicate what Git *head* you want to check out. This also sets the name of your newly checked-out directory, unless you tell it otherwise with *-d <dir-name>*. For example, this checks out *master* branch to the *project-master* directory:

```
cvs co -d project-master master
```

DATABASE BACKEND

git-cvsserver uses one database per Git head (i.e. CVS module) to store information about the repository to maintain consistent CVS revision numbers. The database needs to be updated (i.e. written to) after every commit.

If the commit is done directly by using *git* (as opposed to using *git-cvsserver*) the update will need to happen on the next repository access by *git-cvsserver*, independent of access method and requested operation.

That means that even if you offer only read access (e.g. by using the pserver method), *git-cvsserver* should have write access to the database to work reliably (otherwise you need to make sure that the database is up to date any time *git-cvsserver* is executed).

By default it uses SQLite databases in the Git directory, named *gitcvs.<module-name>.sqlite*. Note that the SQLite backend creates temporary files in the same directory as the database file on write so it might not be enough to grant the users using *git-cvsserver* write access to the database file without granting them write access to the directory, too.

The database cannot be reliably regenerated in a consistent form after the branch it is tracking has changed. Example: For merged branches, *git-cvsserver* only tracks one branch of development, and after a *git merge* an incrementally updated database may track a different branch than a database regenerated from scratch, causing inconsistent CVS revision numbers. *git-cvsserver* has no way of knowing which branch it would have picked if it had been run incrementally pre-merge. So if you have to fully or partially (from old backup) regenerate the database, you should be suspicious of pre-existing CVS sandboxes.

You can configure the database backend with the following configuration variables:

1. Configuring database backend

git-cvsserver uses the Perl DBI module. Please also read its documentation if changing these variables, especially about *DBI->connect()*.

gitcvs.dbName

Database name. The exact meaning depends on the selected database driver, for SQLite this is a filename. Supports variable substitution (see below). May not contain semicolons (*;*). Default: *%Ggitcvs.%m.sqlite*

gitcvs.dbDriver

Used DBI driver. You can specify any available driver for this here, but it might not work. cvsserver is tested with *DBD::SQLite*, reported to work with *DBD::Pg*, and reported **not** to work with *DBD::mysql*. Please regard this as an experimental feature. May not contain colons (*:*). Default: *SQLite*

gitcvs.dbuser

Database user. Only useful if setting *dbDriver*, since SQLite has no concept of database users. Supports variable substitution (see below).

gitcvs.dbPass

Database password. Only useful if setting *dbDriver*, since SQLite has no concept of database passwords.

gitcvs.dbTableNamePrefix

Database table name prefix. Supports variable substitution (see below). Any non-alphabetic characters will be replaced with underscores.

All variables can also be set per access method, see above.

1.1. Variable substitution

In *dbDriver* and *dbUser* you can use the following variables:

%G

Git directory name

%g

Git directory name, where all characters except for alphanumeric ones, ., and - are replaced with _ (this should make it easier to use the directory name in a filename if wanted)

%m

CVS module/Git head name

%a

access method (one of "ext" or "pserver")

%u

Name of the user running *git-cvsserver*. If no name can be determined, the numeric uid is used.

ENVIRONMENT

These variables obviate the need for command-line options in some circumstances, allowing easier restricted usage through git-shell.

GIT_CVSSERVER_BASE_PATH

This variable replaces the argument to --base-path.

GIT_CVSSERVER_ROOT

This variable specifies a single directory, replacing the <*directory*>... argument list. The repository still requires the *gitcvs.enabled* config option, unless *--export-all* is specified.

When these environment variables are set, the corresponding command-line arguments may not be used.

ECLIPSE CVS CLIENT NOTES

To get a checkout with the Eclipse CVS client:

1. Select "Create a new project → From CVS checkout"
2. Create a new location. See the notes below for details on how to choose the right protocol.
3. Browse the *modules* available. It will give you a list of the heads in the repository. You will not be able to browse the tree from there. Only the heads.
4. Pick *HEAD* when it asks what branch/tag to check out. Untick the "launch commit wizard" to avoid committing the .project file.

Protocol notes: If you are using anonymous access via pserver, just select that. Those using SSH access should choose the *ext* protocol, and configure *ext* access on the Preferences#Team#CVS#ExtConnection pane. Set

CVS_SERVER to "*git cvsserver*". Note that password support is not good when using *ext*, you will definitely want to have SSH keys setup.

Alternatively, you can just use the non-standard extssh protocol that Eclipse offer. In that case CVS_SERVER is ignored, and you will have to replace the cvs utility on the server with *git-cvsserver* or manipulate your *.bashrc* so that calling *cvs* effectively calls *git-cvsserver*.

CLIENTS KNOWN TO WORK

- CVS 1.12.9 on Debian
- CVS 1.11.17 on MacOSX (from Fink package)
- Eclipse 3.0, 3.1.2 on MacOSX (see Eclipse CVS Client Notes)
- TortoiseCVS

OPERATIONS SUPPORTED

All the operations required for normal use are supported, including checkout, diff, status, update, log, add, remove, commit.

Most CVS command arguments that read CVS tags or revision numbers (typically -r) work, and also support any git refspec (tag, branch, commit ID, etc). However, CVS revision numbers for non-default branches are not well emulated, and cvs log does not show tags or branches at all. (Non-main-branch CVS revision numbers superficially resemble CVS revision numbers, but they actually encode a git commit ID directly, rather than represent the number of revisions since the branch point.)

Note that there are two ways to checkout a particular branch. As described elsewhere on this page, the "module" parameter of cvs checkout is interpreted as a branch name, and it becomes the main branch. It remains the main branch for a given sandbox even if you temporarily make another branch sticky with cvs update -r. Alternatively, the -r argument can indicate some other branch to actually checkout, even though the module is still the "main" branch. Tradeoffs (as currently implemented): Each new "module" creates a new database on disk with a history for the given module, and after the database is created, operations against that main branch are fast. Or alternatively, -r doesn't take any extra disk space, but may be significantly slower for many operations, like cvs update.

If you want to refer to a git refspec that has characters that are not allowed by CVS, you have two options. First, it may just work to supply the git refspec directly to the appropriate CVS -r argument; some CVS clients don't seem to do much sanity checking of the argument. Second, if that fails, you can use a special character escape mechanism that only uses characters that are valid in CVS tags. A sequence of 4 or 5 characters of the form (underscore ("_"), dash ("-"), one or two characters, and dash ("-")) can encode various characters based on the one or two letters: *"s"* for slash ("/"), *"p"* for period ("."), *"u"* for underscore ("_"), or two hexadecimal digits for any byte value at all (typically an ASCII number, or perhaps a part of a UTF-8 encoded character).

Legacy monitoring operations are not supported (edit, watch and related). Exports and tagging (tags and branches) are not supported at this stage.

1. CRLF Line Ending Conversions

By default the server leaves the *-k* mode blank for all files, which causes the CVS client to treat them as a text files, subject to end-of-line conversion on some platforms.

You can make the server use the end-of-line conversion attributes to set the *-k* modes for files by setting the *gitcvs.usecrlfattr* config variable. See Section G.4.2, “gitattributes(5)” for more information about end-of-line conversion.

Alternatively, if *gitcvs.usecrlfattr* config is not enabled or the attributes do not allow automatic detection for a filename, then the server uses the *gitcvs.allBinary* config for the default setting. If *gitcvs.allBinary* is set, then file not otherwise specified will default to *-kb* mode. Otherwise the *-k* mode is left blank. But if *gitcvs.allBinary* is set to "guess", then the correct *-k* mode will be guessed based on the contents of the file.

For best consistency with *cvs*, it is probably best to override the defaults by setting *gitcvs.usecrlfattr* to true, and *gitcvs.allBinary* to "guess".

DEPENDENCIES

git-cvsserver depends on DBD::SQLite.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.38. git-daemon(1)

2024-07-02
Revision History
2024-07-02

NAME

git-daemon - A really simple server for Git repositories

SYNOPSIS

```
git daemon [--verbose] [--syslog] [--export-all]
        [--timeout=<n>] [--init-timeout=<n>] [--max-connections=<n>]
        [--strict-paths] [--base-path=<path>] [--base-path-relaxed]
        [--user-path | --user-path=<path>]
        [--interpolated-path=<pathtemplate>]
        [--reuseaddr] [--detach] [--pid-file=<file>]
        [--enable=<service>] [--disable=<service>]
        [--allow-override=<service>] [--forbid-override=<service>]
        [--access-hook=<path>] [--[no-]informative-errors]
        [--inetd |
         [--listen=<host-or-ipaddr>] [--port=<n>]
         [--user=<user> [--group=<group>]]]
        [--log-destination=(stderr|syslog|none)]
        [<directory>…]
```

DESCRIPTION

A really simple TCP Git daemon that normally listens on port "DEFAULT_GIT_PORT" aka 9418. It waits for a connection asking for a service, and will serve that service if it is enabled.

It verifies that the directory has the magic file "git-daemon-export-ok", and it will refuse to export any Git directory that hasn't explicitly been marked for export this way (unless the *--export-all* parameter is specified). If you pass some directory paths as *git daemon* arguments, the offers are limited to repositories within those directories.

By default, only *upload-pack* service is enabled, which serves *git fetch-pack* and *git ls-remote* clients, which are invoked from *git fetch*, *git pull*, and *git clone*.

This is ideally suited for read-only updates, i.e., pulling from Git repositories.

An *upload-archive* also exists to serve *git archive*.

OPTIONS

--strict-paths

Match paths exactly (i.e. don't allow "/foo/repo" when the real path is "/foo/repo.git" or "/foo/repo/.git") and don't do user-relative paths. *git daemon* will refuse to start when this option is enabled and no directory arguments are provided.

--base-path=<path>

Remap all the path requests as relative to the given path. This is sort of "Git root" - if you run *git daemon* with *--base-path=/srv/git* on example.com, then if you later try to pull *git://example.com/hello.git*, *git daemon* will interpret the path as */srv/git/hello.git*.

--base-path-relaxed

If --base-path is enabled and repo lookup fails, with this option *git daemon* will attempt to lookup without prefixing the base path. This is useful for switching to --base-path usage, while still allowing the old paths.

--interpolated-path=<pathtemplate>

To support virtual hosting, an interpolated path template can be used to dynamically construct alternate paths. The template supports %H for the target hostname as supplied by the client but converted to all lowercase, %CH for the canonical hostname, %IP for the server's IP address, %P for the port number, and %D for the absolute path of the named repository. After interpolation, the path is validated against the directory list.

--export-all

Allow pulling from all directories that look like Git repositories (have the *objects* and *refs* subdirectories), even if they do not have the *git-daemon-export-ok* file.

--inetd

Have the server run as an inetd service. Implies --syslog (may be overridden with *--log-destination=*). Incompatible with --detach, --port, --listen, --user and --group options.

--listen=<host-or-ipaddr>

Listen on a specific IP address or hostname. IP addresses can be either an IPv4 address or an IPv6 address if supported. If IPv6 is not supported, then --listen=<hostname> is also not supported and --listen must be given an IPv4 address. Can be given more than once. Incompatible with *--inetd* option.

--port=<n>

Listen on an alternative port. Incompatible with *--inetd* option.

--init-timeout=<n>

Timeout (in seconds) between the moment the connection is established and the client request is received (typically a rather low value, since that should be basically immediate).

--timeout=<n>

Timeout (in seconds) for specific client sub-requests. This includes the time it takes for the server to process the sub-request and the time spent waiting for the next client's request.

--max-connections=<n>

Maximum number of concurrent clients, defaults to 32. Set it to zero for no limit.

--syslog

Short for *--log-destination=syslog*.

--log-destination=<destination>

Send log messages to the specified destination. Note that this option does not imply --verbose, thus by default only error conditions will be logged. The <destination> must be one of:

stderr

Write to standard error. Note that if *--detach* is specified, the process disconnects from the real standard error, making this destination effectively equivalent to *none*.

syslog

Write to syslog, using the *git-daemon* identifier.

none

Disable all logging.

The default destination is *syslog* if *--inetd* or *--detach* is specified, otherwise *stderr*.

--user-path , --user-path=<path>

Allow ~user notation to be used in requests. When specified with no parameter, a request to git://host/~alice/foo is taken as a request to access *foo* repository in the home directory of user *alice*. If *--user-path=<path>* is specified, the same request is taken as a request to access *<path>/foo* repository in the home directory of user *alice*.

--verbose

Log details about the incoming connections and requested files.

--reuseaddr

Use SO_REUSEADDR when binding the listening socket. This allows the server to restart without waiting for old connections to time out.

--detach

Detach from the shell. Implies --syslog.

--pid-file=<file>

Save the process id in *file*. Ignored when the daemon is run under *--inetd*.

--user=<user> , --group=<group>

Change daemon's uid and gid before entering the service loop. When only *--user* is given without *--group*, the primary group ID for the user is used. The values of the option are given to *getpwnam(3)* and *getgrnam(3)* and numeric IDs are not supported.

Giving these options is an error when used with *--inetd*; use the facility of inet daemon to achieve the same before spawning *git daemon* if needed.

Like many programs that switch user id, the daemon does not reset environment variables such as *$HOME* when it runs git programs, e.g. *upload-pack* and *receive-pack*. When using this option, you may also want to set and export *HOME* to point at the home directory of *<user>* before starting the daemon, and make sure any Git configuration files in that directory are readable by *<user>*.

--enable=<service> , --disable=<service>

Enable/disable the service site-wide per default. Note that a service disabled site-wide can still be enabled per repository if it is marked overridable and the repository enables the service with a configuration item.

--allow-override=<service> , --forbid-override=<service>

Allow/forbid overriding the site-wide default with per repository configuration. By default, all the services may be overridden.

--[no-]informative-errors

When informative errors are turned on, git-daemon will report more verbose errors to the client, differentiating conditions like "no such repository" from "repository not exported". This is more convenient for clients, but may leak information about the existence of unexported repositories. When informative errors are not enabled, all errors report "access denied" to the client. The default is --no-informative-errors.

--access-hook=<path>

Every time a client connects, first run an external command specified by the <path> with service name (e.g. "upload-pack"), path to the repository, hostname (%H), canonical hostname (%CH), IP address (%IP), and TCP port (%P) as its command-line arguments. The external command can decide to decline the service by exiting with a non-zero status (or to allow it by exiting with a zero status). It can also look at the $REMOTE_ADDR and *$REMOTE_PORT* environment variables to learn about the requestor when making this decision.

The external command can optionally write a single line to its standard output to be sent to the requestor as an error message when it declines the service.

<directory>

The remaining arguments provide a list of directories. If any directories are specified, then the *git-daemon* process will serve a requested directory only if it is contained in one of these directories. If *--strict-paths* is specified, then the requested directory must match one of these directories exactly.

SERVICES

These services can be globally enabled/disabled using the command-line options of this command. If finer-grained control is desired (e.g. to allow *git archive* to be run against only in a few selected repositories the daemon serves), the per-repository configuration file can be used to enable or disable them.

upload-pack

This serves *git fetch-pack* and *git ls-remote* clients. It is enabled by default, but a repository can disable it by setting *daemon.uploadpack* configuration item to *false*.

upload-archive

This serves *git archive --remote*. It is disabled by default, but a repository can enable it by setting *daemon.uploadarch* configuration item to *true*.

receive-pack

This serves *git send-pack* clients, allowing anonymous push. It is disabled by default, as there is *no* authentication in the protocol (in other words, anybody can push anything into the repository, including removal of refs). This is solely meant for a closed LAN setting where everybody is friendly. This service can be enabled by setting *daemon.receivepack* configuration item to *true*.

EXAMPLES

We assume the following in /etc/services

```
$ grep 9418 /etc/services
git             9418/tcp                # Git Version Control System
```

git daemon as inetd server

To set up *git daemon* as an inetd service that handles any repository within */pub/foo* or */pub/bar*, place an entry like the following into */etc/inetd* all on one line:

```
git stream tcp nowait nobody  /usr/bin/git
        git daemon --inetd --verbose --export-all
```

```
                /pub/foo /pub/bar
```

git daemon as inetd server for virtual hosts

To set up *git daemon* as an inetd service that handles repositories for different virtual hosts, *www.example.com* and *www.example.org*, place an entry like the following into */etc/inetd* all on one line:

```
git stream tcp nowait nobody /usr/bin/git
        git daemon --inetd --verbose --export-all
        --interpolated-path=/pub/%H%D
        /pub/www.example.org/software
        /pub/www.example.com/software
        /software
```

In this example, the root-level directory */pub* will contain a subdirectory for each virtual host name supported. Further, both hosts advertise repositories simply as *git://www.example.com/software/repo.git*. For pre-1.4.0 clients, a symlink from */software* into the appropriate default repository could be made as well.

git daemon as regular daemon for virtual hosts

To set up *git daemon* as a regular, non-inetd service that handles repositories for multiple virtual hosts based on their IP addresses, start the daemon like this:

```
git daemon --verbose --export-all
        --interpolated-path=/pub/%IP/%D
        /pub/192.168.1.200/software
        /pub/10.10.220.23/software
```

In this example, the root-level directory */pub* will contain a subdirectory for each virtual host IP address supported. Repositories can still be accessed by hostname though, assuming they correspond to these IP addresses.

selectively enable/disable services per repository

To enable *git archive --remote* and disable *git fetch* against a repository, have the following in the configuration file in the repository (that is the file *config* next to *HEAD*, *refs* and *objects*).

```
[daemon]
        uploadpack = false
        uploadarch = true
```

ENVIRONMENT

git daemon will set REMOTE_ADDR to the IP address of the client that connected to it, if the IP address is available. REMOTE_ADDR will be available in the environment of hooks called when services are performed.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.39. git-describe(1)

2024-07-02

Revision History

2024-07-02

NAME

git-describe - Give an object a human readable name based on an available ref

SYNOPSIS

git describe [--all] [--tags] [--contains] [--abbrev=<n>] [<commit-ish>…]

git describe [--all] [--tags] [--contains] [--abbrev=<n>] --dirty[=<mark>]
git describe <blob>

DESCRIPTION

The command finds the most recent tag that is reachable from a commit. If the tag points to the commit, then only the tag is shown. Otherwise, it suffixes the tag name with the number of additional commits on top of the tagged object and the abbreviated object name of the most recent commit. The result is a "human-readable" object name which can also be used to identify the commit to other git commands.

By default (without --all or --tags) *git describe* only shows annotated tags. For more information about creating annotated tags see the -a and -s options to Section G.3.144, “git-tag(1)”.

If the given object refers to a blob, it will be described as *<commit-ish>:<path>*, such that the blob can be found at *<path>* in the *<commit-ish>*, which itself describes the first commit in which this blob occurs in a reverse revision walk from HEAD.

OPTIONS

<commit-ish>…

Commit-ish object names to describe. Defaults to HEAD if omitted.

--dirty[=<mark>] , --broken[=<mark>]

Describe the state of the working tree. When the working tree matches HEAD, the output is the same as "git describe HEAD". If the working tree has local modification "-dirty" is appended to it. If a repository is corrupt and Git cannot determine if there is local modification, Git will error out, unless --broken is given, which appends the suffix "-broken" instead.

--all

Instead of using only the annotated tags, use any ref found in *refs/* namespace. This option enables matching any known branch, remote-tracking branch, or lightweight tag.

--tags

Instead of using only the annotated tags, use any tag found in *refs/tags* namespace. This option enables matching a lightweight (non-annotated) tag.

--contains

Instead of finding the tag that predates the commit, find the tag that comes after the commit, and thus contains it. Automatically implies --tags.

--abbrev=<n>

Instead of using the default number of hexadecimal digits (which will vary according to the number of objects in the repository with a default of 7) of the abbreviated object name, use <n> digits, or as many digits as needed to form a unique object name. An <n> of 0 will suppress long format, only showing the closest tag.

--candidates=<n>

Instead of considering only the 10 most recent tags as candidates to describe the input commit-ish consider up to <n> candidates. Increasing <n> above 10 will take slightly longer but may produce a more accurate result. An <n> of 0 will cause only exact matches to be output.

--exact-match

Only output exact matches (a tag directly references the supplied commit). This is a synonym for --candidates=0.

--debug

Verbosely display information about the searching strategy being employed to standard error. The tag name will still be printed to standard out.

--long

Always output the long format (the tag, the number of commits and the abbreviated commit name) even when it matches a tag. This is useful when you want to see parts of the commit object name in "describe" output, even when the commit in question happens to be a tagged version. Instead of just emitting the tag name, it will describe such a commit as v1.2-0-gdeadbee (0th commit since tag v1.2 that points at object deadbee….).

--match <pattern>

Only consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix. If used with *--all*, it also considers local branches and remote-tracking references matching the pattern, excluding respectively "refs/heads/" and "refs/remotes/" prefix; references of other types are never considered. If given multiple times, a list of patterns will be accumulated, and tags matching any of the patterns will be considered. Use *--no-match* to clear and reset the list of patterns.

--exclude <pattern>

Do not consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix. If used with *--all*, it also does not consider local branches and remote-tracking references matching the pattern, excluding respectively "refs/heads/" and "refs/remotes/" prefix; references of other types are never considered. If given multiple times, a list of patterns will be accumulated and tags matching any of the patterns will be excluded. When combined with --match a tag will be considered when it matches at least one --match pattern and does not match any of the --exclude patterns. Use *--no-exclude* to clear and reset the list of patterns.

--always

Show uniquely abbreviated commit object as fallback.

--first-parent

Follow only the first parent commit upon seeing a merge commit. This is useful when you wish to not match tags on branches merged in the history of the target commit.

EXAMPLES

With something like git.git current tree, I get:

```
[torvalds@g5 git]$ git describe parent
v1.0.4-14-g2414721
```

i.e. the current head of my "parent" branch is based on v1.0.4, but since it has a few commits on top of that, describe has added the number of additional commits ("14") and an abbreviated object name for the commit itself ("2414721") at the end.

The number of additional commits is the number of commits which would be displayed by "git log v1.0.4..parent". The hash suffix is "-g" + an unambiguous abbreviation for the tip commit of parent (which was *2414721b194453f058079d897d13c4e377f92dc6*). The length of the abbreviation scales as the repository grows, using the approximate number of objects in the repository and a bit of math around the birthday paradox, and defaults to a minimum of 7. The "g" prefix stands for "git" and is used to allow describing the version of a software depending on the SCM the software is managed with. This is useful in an environment where people may use different SCMs.

Doing a *git describe* on a tag-name will just show the tag name:

```
[torvalds@g5 git]$ git describe v1.0.4
v1.0.4
```

With --all, the command can use branch heads as references, so the output shows the reference path as well:

```
[torvalds@g5 git]$ git describe --all --abbrev=4 v1.0.5^2
tags/v1.0.0-21-g975b

[torvalds@g5 git]$ git describe --all --abbrev=4 HEAD^
heads/lt/describe-7-g975b
```

With --abbrev set to 0, the command can be used to find the closest tagname without any suffix:

```
[torvalds@g5 git]$ git describe --abbrev=0 v1.0.5^2
tags/v1.0.0
```

Note that the suffix you get if you type these commands today may be longer than what Linus saw above when he ran these commands, as your Git repository may have new commits whose object names begin with 975b that did not exist back then, and "-g975b" suffix alone may not be sufficient to disambiguate these commits.

SEARCH STRATEGY

For each commit-ish supplied, *git describe* will first look for a tag which tags exactly that commit. Annotated tags will always be preferred over lightweight tags, and tags with newer dates will always be preferred over tags with older dates. If an exact match is found, its name will be output and searching will stop.

If an exact match was not found, *git describe* will walk back through the commit history to locate an ancestor commit which has been tagged. The ancestor's tag will be output along with an abbreviation of the input commit-ish's SHA-1. If *--first-parent* was specified then the walk will only consider the first parent of each commit.

If multiple tags were found during the walk then the tag which has the fewest commits different from the input commit-ish will be selected and output. Here fewest commits different is defined as the number of commits which would be shown by *git log tag..input* will be the smallest number of commits possible.

BUGS

Tree objects as well as tag objects not pointing at commits, cannot be described. When describing blobs, the lightweight tags pointing at blobs are ignored, but the blob is still described as <commit-ish>:<path> despite the lightweight tag being favorable.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.40. git-diagnose(1)

2024-07-02
Revision History
2024-07-02

NAME

git-diagnose - Generate a zip archive of diagnostic information

SYNOPSIS

git diagnose [(-o | --output-directory) <path>] [(-s | --suffix) <format>]
[--mode=<mode>]

DESCRIPTION

Collects detailed information about the user's machine, Git client, and repository state and packages that information into a zip archive. The generated archive can then, for example, be shared with the Git mailing list to help debug an issue or serve as a reference for independent debugging.

By default, the following information is captured in the archive:

- *git version --build-options*
- The path to the repository root
- The available disk space on the filesystem
- The name and size of each packfile, including those in alternate object stores
- The total count of loose objects, as well as counts broken down by *.git/objects* subdirectory

Additional information can be collected by selecting a different diagnostic mode using the *--mode* option.

This tool differs from Section G.3.11, "git-bugreport(1)" in that it collects much more detailed information with a greater focus on reporting the size and data shape of repository contents.

OPTIONS

-o <path> , --output-directory <path>

Place the resulting diagnostics archive in *<path>* instead of the current directory.

-s <format> , --suffix <format>

Specify an alternate suffix for the diagnostics archive name, to create a file named *git-diagnostics-<formatted-suffix>*. This should take the form of a strftime(3) format string; the current local time will be used.

--mode=(stats|all)

Specify the type of diagnostics that should be collected. The default behavior of *git diagnose* is equivalent to *--mode=stats*.

The *--mode=all* option collects everything included in *--mode=stats*, as well as copies of *.git*, *.git/hooks*, *.git/info*, *.git/logs*, and *.git/objects/info* directories. This additional information may be sensitive, as it can be used to reconstruct the full contents of the diagnosed repository. Users should exercise caution when sharing an archive generated with *--mode=all*.

GIT

Part of the Section G.3.1, "git(1)" suite

G.3.41. git-diff-files(1)

2024-07-02
Revision History
2024-07-02

NAME

git-diff-files - Compares files in the working tree and the index

SYNOPSIS

git diff-files [-q] [-0 | -1 | -2 | -3 | -c | --cc] [<common-diff-options>] [<path>…]

DESCRIPTION

Compares the files in the working tree and the index. When paths are specified, compares only those named paths. Otherwise all entries in the index are compared. The output format is the same as for *git diff-index* and *git diff-tree*.

OPTIONS

-p , -u , --patch

Generate patch (see the section called "Generating patch text with -p").

-s , --no-patch

Suppress all output from the diff machinery. Useful for commands like *git show* that show the patch by default to squelch their output, or to cancel the effect of options like *--patch*, *--stat* earlier on the command line in an alias.

-U<n> , --unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies *--patch*.

--output=<file>

Output to a specific file instead of stdout.

--output-indicator-new=<char> , --output-indicator-old=<char> , --output-indicator-context=<char>

Specify the character used to indicate new, old or context lines in the generated patch. Normally they are +, - and ' ' respectively.

--raw

Generate the diff in raw format. This is the default.

--patch-with-raw

Synonym for *-p --raw*.

--indent-heuristic

Enable the heuristic that shifts diff hunk boundaries to make patches easier to read. This is the default.

--no-indent-heuristic

Disable the indent heuristic.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--anchored=<text>

Generate a diff using the "anchored diff" algorithm.

This option may be specified more than once.

If a line exists in both the source and destination, exists only once, and starts with this text, this algorithm attempts to prevent it from appearing as a deletion or addition in the output. It uses the "patience diff" algorithm internally.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

default, myers

The basic greedy diff algorithm. Currently, this is the default.

minimal

Spend extra time to make sure the smallest possible diff is produced.

patience

Use "patience diff" algorithm when generating patches.

histogram

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured the *diff.algorithm* variable to a non-default value and want to use the default one, then you have to use *--diff-algorithm=default* option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by *<width>*. The width of the filename part can be limited by giving another width *<name-width>* after a comma or by setting *diff.statNameWidth=<width>*. The width of the graph part can be limited by using *--stat-graph-width=<width>* or by setting *diff.statGraphWidth=<width>*. Using *--stat* or *--stat-graph-width* affects all commands generating a stat graph, while setting *diff.statNameWidth* or *diff.statGraphWidth* does not affect *git format-patch*. By giving a third parameter *<count>*, you can limit the output to the first *<count>* lines, followed by ... if there are more.

These parameters can also be set individually with *--stat-width=<width>*, *--stat-name-width=<name-width>* and *--stat-count=<count>*.

--compact-summary

Output a condensed summary of extended header information such as file creations or deletions ("new" or "gone", optionally "+l" if it's a symlink) and mode changes ("+x" or "-x" for adding or removing executable bit respectively) in diffstat. The information is put between the filename part and the graph part. Implies *--stat*.

--numstat

Similar to *--stat*, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two - instead of saying *0 0*.

--shortstat

Output only the last line of the *--stat* format containing total number of modified files, as well as number of added and deleted lines.

-X[<param1,param2,…>] , --dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of *--dirstat* can be customized by passing it a comma separated list of parameters. The defaults are controlled by the *diff.dirstat* configuration variable (see Section G.3.29, “git-config(1)”). The following parameters are available:

changes

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

lines

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no nat-

ural concept of lines). This is a more expensive *--dirstat* behavior than the *changes* behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other *--*stat* options.

files

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest *--dirstat* behavior, since it does not have to look at the file contents at all.

cumulative

Count changes in a child directory for the parent directory as well. Note that when using *cumulative*, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the *noncumulative* parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: *--dirstat=files,10,cumulative*.

--cumulative

Synonym for --dirstat=cumulative

--dirstat-by-file[=<param1,param2>…]

Synonym for --dirstat=files,<param1>,<param2>…

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for *-p --stat*.

-z

When *--raw*, *--numstat*, *--name-only* or *--name-status* has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

--name-only

Show only the name of each changed file in the post-image tree. The file names are often encoded in UTF-8. For more information see the discussion about encoding in the Section G.3.74, “git-log(1)” manual page.

--name-status

Show only the name(s) and status of each changed file. See the description of the *--diff-filter* option on what the status letters mean. Just like *--name-only* the file names are often encoded in UTF-8.

--submodule[=<format>]

Specify how differences in submodules are shown. When specifying *--submodule=short* the *short* format is used. This format just shows the names of the commits at the beginning and end of the range. When --

submodule or *--submodule=log* is specified, the *log* format is used. This format lists the commits in the range like Section G.3.141, "git-submodule(1)" *summary* does. When *--submodule=diff* is specified, the *diff* format is used. This format shows an inline diff of the changes in the submodule contents between the commit range. Defaults to *diff.submodule* or the *short* format if the config option is unset.

--color[=<when>]

Show colored diff. *--color* (i.e. without =*<when>*) is the same as *--color=always*. *<when>* can be one of *always*, *never*, or *auto*.

--no-color

Turn off colored diff. It is the same as *--color=never*.

--color-moved[=<mode>]

Moved lines of code are colored differently. The <mode> defaults to *no* if the option is not given and to *zebra* if the option with no mode is given. The mode must be one of:

no

Moved lines are not highlighted.

default

Is a synonym for *zebra*. This may change to a more sensible mode in the future.

plain

Any line that is added in one location and was removed in another location will be colored with *color.diff.newMoved*. Similarly *color.diff.oldMoved* will be used for removed lines that are added somewhere else in the diff. This mode picks up any moved line, but it is not very useful in a review to determine if a block of code was moved without permutation.

blocks

Blocks of moved text of at least 20 alphanumeric characters are detected greedily. The detected blocks are painted using either the *color.diff.{old,new}Moved* color. Adjacent blocks cannot be told apart.

zebra

Blocks of moved text are detected as in *blocks* mode. The blocks are painted using either the *color.diff.{old,new}Moved* color or *color.diff.{old,new}MovedAlternative*. The change between the two colors indicates that a new block was detected.

dimmed-zebra

Similar to *zebra*, but additional dimming of uninteresting parts of moved code is performed. The bordering lines of two adjacent blocks are considered interesting, the rest is uninteresting. *dimmed_zebra* is a deprecated synonym.

--no-color-moved

Turn off move detection. This can be used to override configuration settings. It is the same as *--color-moved=no*.

--color-moved-ws=<modes>

This configures how whitespace is ignored when performing the move detection for *--color-moved*. These modes can be given as a comma separated list:

no

Do not ignore whitespace when performing move detection.

ignore-space-at-eol

Ignore changes in whitespace at EOL.

ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

allow-indentation-change

Initially ignore any whitespace in the move detection, then group the moved code blocks only into a block if the change in whitespace is the same per line. This is incompatible with the other modes.

--no-color-moved-ws

Do not ignore whitespace when performing move detection. This can be used to override configuration settings. It is the same as *--color-moved-ws=no*.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see *--word-diff-regex* below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies *--color*.

plain

Show words as *[-removed-]* and *{+added+}*. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a +/-/` ` character at the beginning of the line and extending to the end of the line. Newlines in the input are represented by a tilde ~ on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies *--word-diff* unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append *|[^[:space:]]*

to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, *--word-diff-regex=*. will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see Section G.4.2, “gitattributes(5)” or Section G.3.29, “git-config(1)”. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to *--word-diff=color* plus (if a regex was specified) *--word-diff-regex=<regex>*.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--[no-]rename-empty

Whether to use empty blobs as rename source.

--check

Warn if changes introduce conflict markers or whitespace errors. What are considered whitespace errors is controlled by *core.whitespace* configuration. By default, trailing whitespaces (including lines that consist solely of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors in the *context*, *old* or *new* lines of the diff. Multiple values are separated by comma, *none* resets previous values, *default* reset the list to *new* and *all* is a shorthand for *old,new,context*. When this option is not given, and the configuration variable *diff.wsErrorHighlight* is not set, only whitespace errors in *new* lines are highlighted. The whitespace errors are colored with *color.diff.whitespace*.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to *--full-index*, output a binary diff that can be applied with *git-apply*. Implies *--patch*.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show the shortest prefix that is at least *<n>* hexdigits long that uniquely refers the object. In diff-patch output format, *--full-index* takes higher precedence, i.e. if *--full-index* is specified, full blob names will be shown regardless of *--abbrev*. Non default number of digits can be specified with *--abbrev=<n>*.

-B[<n>][/<m>] , --break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything new, and the number *m* controls this aspect of the -B option (defaults to 60%). *-B/70%* specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number *n* controls this aspect of the -B option (defaults to 50%). *-B20%* specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>] , --find-renames[=<n>]

Detect renames. If *n* is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, *-M90%* means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a *%* sign, the number is to be read as a fraction, with a decimal point before it. I.e., *-M5* becomes 0.5, and is thus the same as *-M50%*. Similarly, *-M05* is the same as *-M5%*. To limit detection to exact renames, use *-M100%*. The default similarity index is 50%.

-C[<n>] , --find-copies[=<n>]

Detect copies as well as renames. See also *--find-copies-harder*. If *n* is specified, it has the same meaning as for *-M<n>*.

--find-copies-harder

For performance reasons, by default, *-C* option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one *-C* option has the same effect.

-D , --irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and */dev/null*. The resulting patch is not meant to be applied with *patch* or *git apply*; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lacks enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with *-B*, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The *-M* and *-C* options involve some preliminary steps that can detect subsets of renames/copies cheaply, followed by an exhaustive fallback portion that compares all remaining unpaired destinations to all relevant sources. (For renames, only remaining unpaired sources are relevant; for copies, all original sources are relevant.) For N sources and destinations, this exhaustive check is O(N^2). This option prevents the exhaustive portion of rename/copy detection from running if the number of source/destination files involved exceeds the specified number. Defaults to diff.renameLimit. Note that a value of 0 is treated as unlimited.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (*A*), Copied (*C*), Deleted (*D*), Modified (*M*), Renamed (*R*), have their type (i.e. regular file, symlink, submodule, …) changed (*T*), are Unmerged (*U*), are Unknown (*X*), or have had their pairing Broken (*B*). Any combination of the filter characters (including none) can be used. When * (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

Also, these upper-case letters can be downcased to exclude. E.g. *--diff-filter=ad* excludes added and deleted paths.

Note that not all diffs can feature all types. For instance, copied and renamed entries cannot appear if detection for those types is disabled.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter's use.

It is useful when you're looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into *-S*, and keep going until you get the very first version of the block.

Binary files are searched as well.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between *-S<regex> --pickaxe-regex* and *-G<regex>*, consider a commit with the following diff in the same file:

```
+    return frotz(nitfol, two->ptr, 1, 0);
...
-    hit = frotz(nitfol, mf2.ptr, 1, 0);
```

While *git log -G"frotz\(nitfol"* will show this commit, *git log -S"frotz\(nitfol" --pickaxe-regex* will not (because the number of occurrences of that string did not change).

Unless *--text* is supplied patches of binary files without a textconv filter will be ignored.

See the *pickaxe* entry in Section G.4.4, “gitdiffcore(7)” for more information.

--find-object=<object-id>

Look for differences that change the number of occurrences of the specified object. Similar to *-S*, just the argument is different in that it doesn't search for a specific string but for a specific object id.

The object can be a blob or a submodule commit. It implies the *-t* option in *git-log* to also find trees.

--pickaxe-all

When *-S* or *-G* finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to *-S* as an extended POSIX regular expression to match.

-O<orderfile>

Control the order in which files appear in the output. This overrides the *diff.orderFile* configuration variable (see Section G.3.29, “git-config(1)”). To cancel *diff.orderFile*, use *-O/dev/null*.

The output order is determined by the order of glob patterns in <orderfile>. All files with pathnames that match the first pattern are output first, all files with pathnames that match the second pattern (but not the first) are output next, and so on. All files with pathnames that do not match any pattern are output last, as if there was an implicit match-all pattern at the end of the file. If multiple pathnames have the same rank (they match the same pattern but no earlier patterns), their output order relative to each other is the normal order.

<orderfile> is parsed as follows:

- Blank lines are ignored, so they can be used as separators for readability.
- Lines starting with a hash ("#") are ignored, so they can be used for comments. Add a backslash ("\") to the beginning of the pattern if it starts with a hash.
- Each other line contains a single pattern.

Patterns have the same syntax and semantics as patterns used for fnmatch(3) without the FNM_PATHNAME flag, except a pathname also matches a pattern if removing any number of the final pathname components

matches the pattern. For example, the pattern "*foo*bar*" matches "*fooasdfbar*" and "*foo/bar/baz/asdf*" but not "*foobarx*".

--skip-to=<file> , --rotate-to=<file>

Discard the files before the named <file> from the output (i.e. *skip to*), or move them to the end of the output (i.e. *rotate to*). These options were invented primarily for the use of the *git difftool* command, and may not be very useful otherwise.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>] , --no-relative

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument. *--no-relative* can be used to countermand both *diff.relative* config option and previous *--relative*.

-a , --text

Treat all files as text.

--ignore-cr-at-eol

Ignore carriage-return at the end of line when doing a comparison.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b , --ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w , --ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

-I<regex> , --ignore-matching-lines=<regex>

Ignore changes whose all lines match <regex>. This option may be specified more than once.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other. Defaults to *diff.interHunkContext* or 0 if the config option is unset.

-W , --function-context

Show whole function as context lines for each change. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

--exit-code

Make the program exit with codes similar to diff(1). That is, it exits with 1 if there were differences and 0 means no differences.

--quiet

Disable all output of the program. Implies *--exit-code*.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with Section G.4.2, “gitattributes(5)”, you need to use this option with Section G.3.74, “git-log(1)” and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv , --no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See Section G.4.2, “gitattributes(5)” for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for Section G.3.44, “git-diff(1)” and Section G.3.74, “git-log(1)”, but not for Section G.3.54, “git-format-patch(1)” or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in Section G.3.29, “git-config(1)” or Section G.4.10, “gitmodules(5)”. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

--default-prefix

Use the default source and destination prefixes ("a/" and "b/"). This overrides configuration variables such as *diff.noprefix*, *diff.srcPrefix*, *diff.dstPrefix*, and *diff.mnemonicPrefix* (see *git-config*(1)).

--line-prefix=<prefix>

Prepend an additional prefix to every line of output.

--ita-invisible-in-index

By default entries added by "git add -N" appear as an existing empty file in "git diff" and a new file in "git diff --cached". This option makes the entry appear as a new file in "git diff" and non-existent in "git diff --cached". This option could be reverted with *--ita-visible-in-index*. Both options are experimental and could be removed in future.

For more detailed explanation on these common options, see also Section G.4.4, “gitdiffcore(7)”.

-1 --base , -2 --ours , -3 --theirs , -0

Diff against the "base" version, "our branch", or "their branch" respectively. With these options, diffs for merged entries are not shown.

The default is to diff against our branch (-2) and the cleanly resolved paths. The option -0 can be given to omit diff output for unmerged entries and just show "Unmerged".

-c , --cc

This compares stage 2 (our branch), stage 3 (their branch), and the working tree file and outputs a combined diff, similar to the way *diff-tree* shows a merge commit with these flags.

-q

Remain silent even for nonexistent files

Raw output format

The raw output format from "git-diff-index", "git-diff-tree", "git-diff-files" and "git diff --raw" are very similar.

These commands all compare two sets of things; what is compared differs:

git-diff-index <tree-ish>

compares the <tree-ish> and the files on the filesystem.

git-diff-index --cached <tree-ish>

compares the <tree-ish> and the index.

git-diff-tree [-r] <tree-ish-1> <tree-ish-2> [<pattern>…]

compares the trees named by the two arguments.

git-diff-files [<pattern>…]

compares the index and the files on the filesystem.

The "git-diff-tree" command begins its output by printing the hash of what is being compared. After that, all the commands print one output line per changed file.

An output line is formatted this way:

```
in-place edit  :100644 100644 bcd1234 0123456 M file0
copy-edit      :100644 100644 abcd123 1234567 C68 file1 file2
rename-edit    :100644 100644 abcd123 1234567 R86 file1 file3
create         :000000 100644 0000000 1234567 A file4
delete         :100644 000000 1234567 0000000 D file5
unmerged       :000000 000000 0000000 0000000 U file6
```

That is, from the left to the right:

1. a colon.
2. mode for "src"; 000000 if creation or unmerged.
3. a space.
4. mode for "dst"; 000000 if deletion or unmerged.
5. a space.
6. sha1 for "src"; 0{40} if creation or unmerged.

7. a space.
8. sha1 for "dst"; 0{40} if deletion, unmerged or "work tree out of sync with the index".
9. a space.
10. status, followed by optional "score" number.
11. a tab or a NUL when *-z* option is used.
12. path for "src"
13. a tab or a NUL when *-z* option is used; only exists for C or R.
14. path for "dst"; only exists for C or R.
15. an LF or a NUL when *-z* option is used, to terminate the record.

Possible status letters are:

- A: addition of a file
- C: copy of a file into a new one
- D: deletion of a file
- M: modification of the contents or mode of a file
- R: renaming of a file
- T: change in the type of the file (regular file, symbolic link or submodule)
- U: file is unmerged (you must complete the merge before it can be committed)
- X: "unknown" change type (most probably a bug, please report it)

Status letters C and R are always followed by a score (denoting the percentage of similarity between the source and target of the move or copy). Status letter M may be followed by a score (denoting the percentage of dissimilarity) for file rewrites.

The sha1 for "dst" is shown as all 0's if a file on the filesystem is out of sync with the index.

Example:

```
:100644 100644 5be4a4a 0000000 M file.c
```

Without the *-z* option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). Using *-z* the filename is output verbatim and the line is terminated by a NUL byte.

diff format for merges

"git-diff-tree", "git-diff-files" and "git-diff --raw" can take *-c* or *--cc* option to generate diff output also for merge commits. The output differs from the format described above in the following way:

1. there is a colon for each parent
2. there are more "src" modes and "src" sha1
3. status is concatenated status characters for each parent
4. no optional "score" number
5. tab-separated pathname(s) of the file

For *-c* and *--cc*, only the destination or final path is shown even if the file was renamed on any side of history. With *--combined-all-paths*, the name of the path in each parent is shown followed by the name of the path in the merge commit.

Examples for *-c* and *--cc* without *--combined-all-paths*:

```
::100644 100644 100644 fabadb8 cc95eb0 4866510 MM       desc.c
::100755 100755 100755 52b7a2d 6d1ac04 d2ac7d7 RM       bar.sh
::100644 100644 100644 e07d6c5 9042e82 ee91881 RR       phooey.c
```

Examples when *--combined-all-paths* added to either *-c* or *--cc*:

```
::100644 100644 100644 fabadb8 cc95eb0 4866510 MM       desc.c  desc.c
 desc.c
::100755 100755 100755 52b7a2d 6d1ac04 d2ac7d7 RM       foo.sh  bar.sh
 bar.sh
::100644 100644 100644 e07d6c5 9042e82 ee91881 RR       fooey.c fuey.c
 phooey.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patch text with -p

Running Section G.3.44, “git-diff(1)”, Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, Section G.3.42, “git-diff-index(1)”, Section G.3.43, “git-diff-tree(1)”, or Section G.3.41, “git-diff-files(1)” with the *-p* option produces patch text. You can customize the creation of patch text via the *GIT_EXTERNAL_DIFF* and the *GIT_DIFF_OPTS* environment variables (see Section G.3.1, “git(1)”), and the *diff* attribute (see Section G.4.2, “gitattributes(5)”).

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded by a "git diff" header that looks like this:

   ```
   diff --git a/file1 b/file2
   ```

   The *a/* and *b/* filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, */dev/null* is *not* used in place of the *a/* or *b/* filenames.

   When a rename/copy is involved, *file1* and *file2* show the name of the source file of the rename/copy and the name of the file that the rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

   ```
   old mode <mode>
   new mode <mode>
   deleted file mode <mode>
   new file mode <mode>
   copy from <path>
   copy to <path>
   rename from <path>
   rename to <path>
   similarity index <number>
   dissimilarity index <number>
   index <hash>..<hash> <mode>
   ```

   File modes are printed as 6-digit octal numbers including the file type and file permission bits.

   Path names in extended headers do not include the *a/* and *b/* prefixes.

   The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100%

is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the blob object names before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. Pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

4. All the *file1* files in the output refer to files before the commit, and all the *file2* files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

5. Hunk headers mention the name of the function to which the hunk applies. See "Defining a custom hunk-header" in Section G.4.2, “gitattributes(5)” for details of how to tailor this to specific languages.

Combined diff format

Any diff-generating command can take the *-c* or *--cc* option to produce a *combined diff* when showing a merge. This is the default format when showing merges with Section G.3.44, “git-diff(1)” or Section G.3.134, “git-show(1)”. Note also that you can give suitable *--diff-merges* option to any of these commands to force generation of diffs in a specific format.

A "combined diff" format looks like this:

```
diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
        return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +      unsigned char sha1[20];
 +      struct commit *cmit;
        struct commit_list *list;
        static int initialized = 0;
        struct commit_name *n;

 +      if (get_sha1(arg, sha1) < 0)
 +              usage(describe_usage);
 +      cmit = lookup_commit_reference(sha1);
 +      if (!cmit)
 +              usage(describe_usage);
 +
        if (!initialized) {
                initialized = 1;
                for_each_ref(get_name);
```

1. It is preceded by a "git diff" header, that looks like this (when the *-c* option is used):

```
diff --combined file
```

or like this (when the *--cc* option is used):

```
diff --cc file
```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

```
index <hash>,<hash>..<hash>
mode <mode>,<mode>..<mode>
new file mode <mode>
deleted file mode <mode>,<mode>
```

The *mode* *<mode>*,*<mode>*..*<mode>* line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected content movement (renames and copying detection) are designed to work with the diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by a two-line from-file/to-file header:

```
--- a/file
+++ b/file
```

Similar to the two-line header for the traditional *unified* diff format, */dev/null* is used to signal created or deleted files.

However, if the --combined-all-paths option is provided, instead of a two-line from-file/to-file, you get an N +1 line from-file/to-file header, where N is the number of parents in the merge commit:

```
--- a/file
--- a/file
--- a/file
+++ b/file
```

This extended format can be useful if rename or copy detection is active, to allow you to see the original name of the file in different parents.

4. Chunk header format is modified to prevent people from accidentally feeding it to *patch -p1*. Combined diff format was created for review of merge commit changes, and was not meant to be applied. The change is similar to the change in the extended *index* header:

```
@@@ <from-file-range> <from-file-range> <to-file-range> @@@
```

There are (number of parents + 1) @ characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has - (minus -- appears in A but removed in B), + (plus -- missing in A but added to B), or " " (space -- unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X's line is different from it.

A - character in the column N means that the line appears in fileN but it does not appear in the result. A + character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two - removals from both file1 and file2, plus ++ to mean one line that was added does not appear in either file1 or file2). Also, eight other lines are the same from file1 but do not appear in file2 (hence prefixed with +).

When shown by *git diff-tree -c*, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by *git diff-files -c*, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

other diff formats

The *--summary* option describes newly added, deleted, renamed and copied files. The *--stat* option adds diffstat(1) graph to the output. These options can be combined with other options, such as *-p*, and are meant for human consumption.

When showing a change that involves a rename or a copy, *--stat* output formats the pathnames compactly by combining common prefix and suffix of the pathnames. For example, a change that moves *arch/i386/Makefile* to *arch/x86/Makefile* while modifying 4 lines will be shown like this:

```
arch/{i386 => x86}/Makefile     |    4 +--
```

The *--numstat* option gives the diffstat(1) information but is designed for easier machine consumption. An entry in *--numstat* output looks like this:

```
1       2       README
3       1       arch/{i386 => x86}/Makefile
```

That is, from left to right:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. pathname (possibly with rename/copy information);
6. a newline.

When *-z* output option is in effect, the output is formatted this way:

```
1       2       README NUL
3       1       NUL arch/i386/Makefile NUL arch/x86/Makefile NUL
```

That is:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. a NUL (only exists if renamed/copied);
6. pathname in preimage;
7. a NUL (only exists if renamed/copied);
8. pathname in postimage (only exists if renamed/copied);
9. a NUL.

The extra *NUL* before the preimage path in renamed case is to allow scripts that read the output to tell if the current record being read is a single-path record or a rename/copy record without reading ahead. After reading added and deleted lines, reading up to *NUL* would yield the pathname, but if that is *NUL*, the record will show two paths.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.42. git-diff-index(1)

2024-07-02
Revision History
2024-07-02

NAME

git-diff-index - Compare a tree to the working tree or index

SYNOPSIS

git diff-index [-m] [--cached] [--merge-base] [<common-diff-options>] <tree-ish> [<path>…]

DESCRIPTION

Compare the content and mode of the blobs found in a tree object with the corresponding tracked files in the working tree, or with the corresponding paths in the index. When <path> arguments are present, compare only paths matching those patterns. Otherwise all tracked files are compared.

OPTIONS

-p , -u , --patch

Generate patch (see the section called “Generating patch text with -p”).

-s , --no-patch

Suppress all output from the diff machinery. Useful for commands like *git show* that show the patch by default to squelch their output, or to cancel the effect of options like *--patch*, *--stat* earlier on the command line in an alias.

-U<n> , --unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies *--patch*.

--output=<file>

Output to a specific file instead of stdout.

--output-indicator-new=<char> , --output-indicator-old=<char> , --output-indicator-context=<char>

Specify the character used to indicate new, old or context lines in the generated patch. Normally they are +, - and ' ' respectively.

--raw

Generate the diff in raw format. This is the default.

--patch-with-raw

Synonym for *-p --raw*.

--indent-heuristic

Enable the heuristic that shifts diff hunk boundaries to make patches easier to read. This is the default.

--no-indent-heuristic

Disable the indent heuristic.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--anchored=<text>

Generate a diff using the "anchored diff" algorithm.

This option may be specified more than once.

If a line exists in both the source and destination, exists only once, and starts with this text, this algorithm attempts to prevent it from appearing as a deletion or addition in the output. It uses the "patience diff" algorithm internally.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

default, *myers*

The basic greedy diff algorithm. Currently, this is the default.

minimal

Spend extra time to make sure the smallest possible diff is produced.

patience

Use "patience diff" algorithm when generating patches.

histogram

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured the *diff.algorithm* variable to a non-default value and want to use the default one, then you have to use *--diff-algorithm=default* option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by *<width>*. The width of the filename part can be limited by giving another width *<name-width>* after a comma or by setting *diff.statNameWidth=<width>*. The width of the graph part can be limited by using *--stat-graph-width=<width>* or by setting *diff.statGraphWidth=<width>*. Using *--stat* or *--stat-graph-width* affects all commands generating a stat graph, while setting *diff.statNameWidth* or *diff.statGraphWidth* does not affect *git format-patch*. By giving a third parameter *<count>*, you can limit the output to the first *<count>* lines, followed by ... if there are more.

These parameters can also be set individually with *--stat-width=<width>*, *--stat-name-width=<name-width>* and *--stat-count=<count>*.

--compact-summary

Output a condensed summary of extended header information such as file creations or deletions ("new" or "gone", optionally "+l" if it's a symlink) and mode changes ("+x" or "-x" for adding or removing executable bit respectively) in diffstat. The information is put between the filename part and the graph part. Implies *--stat*.

--numstat

Similar to --*stat*, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two - instead of saying *0 0*.

--shortstat

Output only the last line of the --*stat* format containing total number of modified files, as well as number of added and deleted lines.

-X[<param1,param2,…>] , --dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of --*dirstat* can be customized by passing it a comma separated list of parameters. The defaults are controlled by the *diff.dirstat* configuration variable (see Section G.3.29, "git-config(1)"). The following parameters are available:

changes

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

lines

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive --*dirstat* behavior than the *changes* behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other --**stat* options.

files

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest --*dirstat* behavior, since it does not have to look at the file contents at all.

cumulative

Count changes in a child directory for the parent directory as well. Note that when using *cumulative*, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the *noncumulative* parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: --*dirstat=files,10,cumulative*.

--cumulative

Synonym for --dirstat=cumulative

--dirstat-by-file[=<param1,param2>…]

Synonym for --dirstat=files,<param1>,<param2>…

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for *-p --stat*.

-z

When *--raw*, *--numstat*, *--name-only* or *--name-status* has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

--name-only

Show only the name of each changed file in the post-image tree. The file names are often encoded in UTF-8. For more information see the discussion about encoding in the Section G.3.74, “git-log(1)” manual page.

--name-status

Show only the name(s) and status of each changed file. See the description of the *--diff-filter* option on what the status letters mean. Just like *--name-only* the file names are often encoded in UTF-8.

--submodule[=<format>]

Specify how differences in submodules are shown. When specifying *--submodule=short* the *short* format is used. This format just shows the names of the commits at the beginning and end of the range. When *--submodule* or *--submodule=log* is specified, the *log* format is used. This format lists the commits in the range like Section G.3.141, “git-submodule(1)” *summary* does. When *--submodule=diff* is specified, the *diff* format is used. This format shows an inline diff of the changes in the submodule contents between the commit range. Defaults to *diff.submodule* or the *short* format if the config option is unset.

--color[=<when>]

Show colored diff. *--color* (i.e. without =<*when*>) is the same as *--color=always*. <*when*> can be one of *always*, *never*, or *auto*.

--no-color

Turn off colored diff. It is the same as *--color=never*.

--color-moved[=<mode>]

Moved lines of code are colored differently. The <mode> defaults to *no* if the option is not given and to *zebra* if the option with no mode is given. The mode must be one of:

no

Moved lines are not highlighted.

default

Is a synonym for *zebra*. This may change to a more sensible mode in the future.

plain

Any line that is added in one location and was removed in another location will be colored with *color.diff.newMoved*. Similarly *color.diff.oldMoved* will be used for removed lines that are added somewhere else in the diff. This mode picks up any moved line, but it is not very useful in a review to determine if a block of code was moved without permutation.

blocks

Blocks of moved text of at least 20 alphanumeric characters are detected greedily. The detected blocks are painted using either the *color.diff.{old,new}Moved* color. Adjacent blocks cannot be told apart.

zebra

Blocks of moved text are detected as in *blocks* mode. The blocks are painted using either the *color.diff.{old,new}Moved* color or *color.diff.{old,new}MovedAlternative*. The change between the two colors indicates that a new block was detected.

dimmed-zebra

Similar to *zebra*, but additional dimming of uninteresting parts of moved code is performed. The bordering lines of two adjacent blocks are considered interesting, the rest is uninteresting. *dimmed_zebra* is a deprecated synonym.

--no-color-moved

Turn off move detection. This can be used to override configuration settings. It is the same as *--color-moved=no*.

--color-moved-ws=<modes>

This configures how whitespace is ignored when performing the move detection for *--color-moved*. These modes can be given as a comma separated list:

no

Do not ignore whitespace when performing move detection.

ignore-space-at-eol

Ignore changes in whitespace at EOL.

ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

allow-indentation-change

Initially ignore any whitespace in the move detection, then group the moved code blocks only into a block if the change in whitespace is the same per line. This is incompatible with the other modes.

--no-color-moved-ws

Do not ignore whitespace when performing move detection. This can be used to override configuration settings. It is the same as *--color-moved-ws=no*.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see *--word-diff-regex* below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies *--color*.

plain

Show words as *[-removed-]* and *{+added+}*. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a +/-/` ` character at the beginning of the line and extending to the end of the line. Newlines in the input are represented by a tilde ~ on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies *--word-diff* unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append *|[^[:space:]]* to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, *--word-diff-regex=.* will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see Section G.4.2, “gitattributes(5)” or Section G.3.29, “git-config(1)”. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to *--word-diff=color* plus (if a regex was specified) *--word-diff-regex=<regex>*.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--[no-]rename-empty

Whether to use empty blobs as rename source.

--check

Warn if changes introduce conflict markers or whitespace errors. What are considered whitespace errors is controlled by *core.whitespace* configuration. By default, trailing whitespaces (including lines that consist solely of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors in the *context*, *old* or *new* lines of the diff. Multiple values are separated by comma, *none* resets previous values, *default* reset the list to *new* and *all* is a shorthand for *old,new,context*. When this option is not given, and the configuration variable *diff.wsErrorHighlight* is not set, only whitespace errors in *new* lines are highlighted. The whitespace errors are colored with *color.diff.whitespace*.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to *--full-index*, output a binary diff that can be applied with *git-apply*. Implies *--patch*.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show the shortest prefix that is at least *<n>* hexdigits long that uniquely refers the object. In diff-patch output format, *--full-index* takes higher precedence, i.e. if *--full-index* is specified, full blob names will be shown regardless of *--abbrev*. Non default number of digits can be specified with *--abbrev=<n>*.

-B[<n>][/<m>] , --break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything new, and the number *m* controls this aspect of the -B option (defaults to 60%). *-B/70%* specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number *n* controls this aspect of the -B option (defaults to 50%). *-B20%* specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>] , --find-renames[=<n>]

Detect renames. If *n* is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, *-M90%* means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a *%* sign, the number is to be read as a fraction, with a decimal point before it. I.e., *-M5* becomes 0.5, and is thus the same as *-M50%*. Similarly, *-M05* is the same as *-M5%*. To limit detection to exact renames, use *-M100%*. The default similarity index is 50%.

-C[<n>] , --find-copies[=<n>]

Detect copies as well as renames. See also *--find-copies-harder*. If *n* is specified, it has the same meaning as for *-M<n>*.

--find-copies-harder

For performance reasons, by default, *-C* option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one *-C* option has the same effect.

-D , --irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and */dev/null*. The resulting patch is not meant to be applied with *patch* or *git apply*; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lacks enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with *-B*, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The *-M* and *-C* options involve some preliminary steps that can detect subsets of renames/copies cheaply, followed by an exhaustive fallback portion that compares all remaining unpaired destinations to all relevant sources. (For renames, only remaining unpaired sources are relevant; for copies, all original sources are rele-

vant.) For N sources and destinations, this exhaustive check is O(N^2). This option prevents the exhaustive portion of rename/copy detection from running if the number of source/destination files involved exceeds the specified number. Defaults to diff.renameLimit. Note that a value of 0 is treated as unlimited.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (*A*), Copied (*C*), Deleted (*D*), Modified (*M*), Renamed (*R*), have their type (i.e. regular file, symlink, submodule, …) changed (*T*), are Unmerged (*U*), are Unknown (*X*), or have had their pairing Broken (*B*). Any combination of the filter characters (including none) can be used. When * (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

Also, these upper-case letters can be downcased to exclude. E.g. *--diff-filter=ad* excludes added and deleted paths.

Note that not all diffs can feature all types. For instance, copied and renamed entries cannot appear if detection for those types is disabled.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter's use.

It is useful when you're looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into *-S*, and keep going until you get the very first version of the block.

Binary files are searched as well.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between *-S<regex> --pickaxe-regex* and *-G<regex>*, consider a commit with the following diff in the same file:

```
+    return frotz(nitfol, two->ptr, 1, 0);
...
-    hit = frotz(nitfol, mf2.ptr, 1, 0);
```

While *git log -G"frotz\(nitfol"* will show this commit, *git log -S"frotz\(nitfol" --pickaxe-regex* will not (because the number of occurrences of that string did not change).

Unless *--text* is supplied patches of binary files without a textconv filter will be ignored.

See the *pickaxe* entry in Section G.4.4, “gitdiffcore(7)” for more information.

--find-object=<object-id>

Look for differences that change the number of occurrences of the specified object. Similar to *-S*, just the argument is different in that it doesn't search for a specific string but for a specific object id.

The object can be a blob or a submodule commit. It implies the *-t* option in *git-log* to also find trees.

--pickaxe-all

When *-S* or *-G* finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to *-S* as an extended POSIX regular expression to match.

-O<orderfile>

Control the order in which files appear in the output. This overrides the *diff.orderFile* configuration variable (see Section G.3.29, “git-config(1)”). To cancel *diff.orderFile*, use *-O/dev/null*.

The output order is determined by the order of glob patterns in <orderfile>. All files with pathnames that match the first pattern are output first, all files with pathnames that match the second pattern (but not the first) are output next, and so on. All files with pathnames that do not match any pattern are output last, as if there was an implicit match-all pattern at the end of the file. If multiple pathnames have the same rank (they match the same pattern but no earlier patterns), their output order relative to each other is the normal order.

<orderfile> is parsed as follows:

- Blank lines are ignored, so they can be used as separators for readability.
- Lines starting with a hash ("#") are ignored, so they can be used for comments. Add a backslash ("\") to the beginning of the pattern if it starts with a hash.
- Each other line contains a single pattern.

Patterns have the same syntax and semantics as patterns used for fnmatch(3) without the FNM_PATHNAME flag, except a pathname also matches a pattern if removing any number of the final pathname components matches the pattern. For example, the pattern "*foo*bar*" matches "*fooasdfbar*" and "*foo/bar/baz/asdf*" but not "*foobarx*".

--skip-to=<file> , --rotate-to=<file>

Discard the files before the named <file> from the output (i.e. *skip to*), or move them to the end of the output (i.e. *rotate to*). These options were invented primarily for the use of the *git difftool* command, and may not be very useful otherwise.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>] , --no-relative

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument. *--no-relative* can be used to countermand both *diff.relative* config option and previous *--relative*.

-a , --text

Treat all files as text.

--ignore-cr-at-eol

Ignore carriage-return at the end of line when doing a comparison.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b , --ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w , --ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

-I<regex> , --ignore-matching-lines=<regex>

Ignore changes whose all lines match <regex>. This option may be specified more than once.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other. Defaults to *diff.interHunkContext* or 0 if the config option is unset.

-W , --function-context

Show whole function as context lines for each change. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

--exit-code

Make the program exit with codes similar to diff(1). That is, it exits with 1 if there were differences and 0 means no differences.

--quiet

Disable all output of the program. Implies *--exit-code*.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with Section G.4.2, “gitattributes(5)”, you need to use this option with Section G.3.74, “git-log(1)” and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv , --no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See Section G.4.2, “gitattributes(5)” for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for Section G.3.44, “git-diff(1)” and Section G.3.74, “git-log(1)”, but not for Section G.3.54, “git-format-patch(1)” or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in Section G.3.29, “git-config(1)” or Section G.4.10, “gitmodules(5)”. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

--default-prefix

Use the default source and destination prefixes ("a/" and "b/"). This overrides configuration variables such as *diff.noprefix*, *diff.srcPrefix*, *diff.dstPrefix*, and *diff.mnemonicPrefix* (see *git-config*(1)).

--line-prefix=<prefix>

Prepend an additional prefix to every line of output.

--ita-invisible-in-index

By default entries added by "git add -N" appear as an existing empty file in "git diff" and a new file in "git diff --cached". This option makes the entry appear as a new file in "git diff" and non-existent in "git diff --cached". This option could be reverted with *--ita-visible-in-index*. Both options are experimental and could be removed in future.

For more detailed explanation on these common options, see also Section G.4.4, “gitdiffcore(7)”.

<tree-ish>

The id of a tree object to diff against.

--cached

Do not consider the on-disk file at all.

--merge-base

Instead of comparing <tree-ish> directly, use the merge base between <tree-ish> and HEAD instead. <tree-ish> must be a commit.

-m

By default, files recorded in the index but not checked out are reported as deleted. This flag makes *git diff-index* say that all non-checked-out files are up to date.

Raw output format

The raw output format from "git-diff-index", "git-diff-tree", "git-diff-files" and "git diff --raw" are very similar.

These commands all compare two sets of things; what is compared differs:

git-diff-index <tree-ish>

compares the <tree-ish> and the files on the filesystem.

git-diff-index --cached <tree-ish>

compares the <tree-ish> and the index.

git-diff-tree [-r] <tree-ish-1> <tree-ish-2> [<pattern>…]

compares the trees named by the two arguments.

git-diff-files [<pattern>…]

compares the index and the files on the filesystem.

The "git-diff-tree" command begins its output by printing the hash of what is being compared. After that, all the commands print one output line per changed file.

An output line is formatted this way:

```
in-place edit  :100644 100644 bcd1234 0123456 M file0
copy-edit      :100644 100644 abcd123 1234567 C68 file1 file2
rename-edit    :100644 100644 abcd123 1234567 R86 file1 file3
create         :000000 100644 0000000 1234567 A file4
delete         :100644 000000 1234567 0000000 D file5
unmerged       :000000 000000 0000000 0000000 U file6
```

That is, from the left to the right:

1. a colon.
2. mode for "src"; 000000 if creation or unmerged.
3. a space.
4. mode for "dst"; 000000 if deletion or unmerged.
5. a space.
6. sha1 for "src"; 0{40} if creation or unmerged.
7. a space.
8. sha1 for "dst"; 0{40} if deletion, unmerged or "work tree out of sync with the index".
9. a space.
10. status, followed by optional "score" number.
11. a tab or a NUL when *-z* option is used.
12. path for "src"
13. a tab or a NUL when *-z* option is used; only exists for C or R.
14. path for "dst"; only exists for C or R.
15. an LF or a NUL when *-z* option is used, to terminate the record.

Possible status letters are:

- A: addition of a file
- C: copy of a file into a new one
- D: deletion of a file
- M: modification of the contents or mode of a file
- R: renaming of a file
- T: change in the type of the file (regular file, symbolic link or submodule)
- U: file is unmerged (you must complete the merge before it can be committed)
- X: "unknown" change type (most probably a bug, please report it)

Status letters C and R are always followed by a score (denoting the percentage of similarity between the source and target of the move or copy). Status letter M may be followed by a score (denoting the percentage of dissimilarity) for file rewrites.

The sha1 for "dst" is shown as all 0's if a file on the filesystem is out of sync with the index.

Example:

```
:100644 100644 5be4a4a 0000000 M file.c
```

Without the *-z* option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). Using *-z* the filename is output verbatim and the line is terminated by a NUL byte.

diff format for merges

"git-diff-tree", "git-diff-files" and "git-diff --raw" can take *-c* or *--cc* option to generate diff output also for merge commits. The output differs from the format described above in the following way:

1. there is a colon for each parent
2. there are more "src" modes and "src" sha1
3. status is concatenated status characters for each parent
4. no optional "score" number
5. tab-separated pathname(s) of the file

For *-c* and *--cc*, only the destination or final path is shown even if the file was renamed on any side of history. With *--combined-all-paths*, the name of the path in each parent is shown followed by the name of the path in the merge commit.

Examples for *-c* and *--cc* without *--combined-all-paths*:

```
::100644 100644 100644 fabadb8 cc95eb0 4866510 MM       desc.c
::100755 100755 100755 52b7a2d 6d1ac04 d2ac7d7 RM       bar.sh
::100644 100644 100644 e07d6c5 9042e82 ee91881 RR       phooey.c
```

Examples when *--combined-all-paths* added to either *-c* or *--cc*:

```
::100644 100644 100644 fabadb8 cc95eb0 4866510 MM       desc.c  desc.c
 desc.c
::100755 100755 100755 52b7a2d 6d1ac04 d2ac7d7 RM       foo.sh  bar.sh
 bar.sh
::100644 100644 100644 e07d6c5 9042e82 ee91881 RR       fooey.c fuey.c
 phooey.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patch text with -p

Running Section G.3.44, “git-diff(1)”, Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, Section G.3.42, “git-diff-index(1)”, Section G.3.43, “git-diff-tree(1)”, or Section G.3.41, “git-diff-files(1)” with the *-p* option produces patch text. You can customize the creation of patch text via the *GIT_EXTERNAL_DIFF* and the *GIT_DIFF_OPTS* environment variables (see Section G.3.1, “git(1)”), and the *diff* attribute (see Section G.4.2, “gitattributes(5)”).

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded by a "git diff" header that looks like this:

```
diff --git a/file1 b/file2
```

The *a/* and *b/* filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, */dev/null* is *not* used in place of the *a/* or *b/* filenames.

When a rename/copy is involved, *file1* and *file2* show the name of the source file of the rename/copy and the name of the file that the rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

```
old mode <mode>
new mode <mode>
deleted file mode <mode>
new file mode <mode>
copy from <path>
copy to <path>
rename from <path>
rename to <path>
similarity index <number>
dissimilarity index <number>
index <hash>..<hash> <mode>
```

File modes are printed as 6-digit octal numbers including the file type and file permission bits.

Path names in extended headers do not include the *a/* and *b/* prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the blob object names before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. Pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

4. All the *file1* files in the output refer to files before the commit, and all the *file2* files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

5. Hunk headers mention the name of the function to which the hunk applies. See "Defining a custom hunk-header" in Section G.4.2, “gitattributes(5)” for details of how to tailor this to specific languages.

Combined diff format

Any diff-generating command can take the *-c* or *--cc* option to produce a *combined diff* when showing a merge. This is the default format when showing merges with Section G.3.44, “git-diff(1)” or Section G.3.134, “git-show(1)”. Note also that you can give suitable *--diff-merges* option to any of these commands to force generation of diffs in a specific format.

A "combined diff" format looks like this:

```
diff --combined describe.c
```

```
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
        return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +      unsigned char sha1[20];
 +      struct commit *cmit;
        struct commit_list *list;
        static int initialized = 0;
        struct commit_name *n;

 +      if (get_sha1(arg, sha1) < 0)
 +              usage(describe_usage);
 +      cmit = lookup_commit_reference(sha1);
 +      if (!cmit)
 +              usage(describe_usage);
 +
        if (!initialized) {
                initialized = 1;
                for_each_ref(get_name);
```

1. It is preceded by a "git diff" header, that looks like this (when the *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when the *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index <hash>,<hash>..<hash>
   mode <mode>,<mode>..<mode>
   new file mode <mode>
   deleted file mode <mode>,<mode>
   ```

   The *mode <mode>,<mode>..<mode>* line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected content movement (renames and copying detection) are designed to work with the diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by a two-line from-file/to-file header:

   ```
   --- a/file
   +++ b/file
   ```

   Similar to the two-line header for the traditional *unified* diff format, */dev/null* is used to signal created or deleted files.

   However, if the --combined-all-paths option is provided, instead of a two-line from-file/to-file, you get an N +1 line from-file/to-file header, where N is the number of parents in the merge commit:

   ```
   --- a/file
   --- a/file
   --- a/file
   ```

```
+++ b/file
```

This extended format can be useful if rename or copy detection is active, to allow you to see the original name of the file in different parents.

4. Chunk header format is modified to prevent people from accidentally feeding it to *patch -p1*. Combined diff format was created for review of merge commit changes, and was not meant to be applied. The change is similar to the change in the extended *index* header:

```
@@@ <from-file-range> <from-file-range> <to-file-range> @@@
```

There are (number of parents + 1) @ characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has - (minus -- appears in A but removed in B), + (plus -- missing in A but added to B), or " " (space -- unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X's line is different from it.

A - character in the column N means that the line appears in fileN but it does not appear in the result. A + character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two - removals from both file1 and file2, plus ++ to mean one line that was added does not appear in either file1 or file2). Also, eight other lines are the same from file1 but do not appear in file2 (hence prefixed with +).

When shown by *git diff-tree -c*, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by *git diff-files -c*, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

other diff formats

The *--summary* option describes newly added, deleted, renamed and copied files. The *--stat* option adds diffstat(1) graph to the output. These options can be combined with other options, such as *-p*, and are meant for human consumption.

When showing a change that involves a rename or a copy, *--stat* output formats the pathnames compactly by combining common prefix and suffix of the pathnames. For example, a change that moves *arch/i386/Makefile* to *arch/x86/Makefile* while modifying 4 lines will be shown like this:

```
arch/{i386 => x86}/Makefile    |   4 +--
```

The *--numstat* option gives the diffstat(1) information but is designed for easier machine consumption. An entry in *--numstat* output looks like this:

```
1       2       README
3       1       arch/{i386 => x86}/Makefile
```

That is, from left to right:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. pathname (possibly with rename/copy information);
6. a newline.

When *-z* output option is in effect, the output is formatted this way:

```
1       2       README NUL
3       1       NUL arch/i386/Makefile NUL arch/x86/Makefile NUL
```

That is:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. a NUL (only exists if renamed/copied);
6. pathname in preimage;
7. a NUL (only exists if renamed/copied);
8. pathname in postimage (only exists if renamed/copied);
9. a NUL.

The extra *NUL* before the preimage path in renamed case is to allow scripts that read the output to tell if the current record being read is a single-path record or a rename/copy record without reading ahead. After reading added and deleted lines, reading up to *NUL* would yield the pathname, but if that is *NUL*, the record will show two paths.

OPERATING MODES

You can choose whether you want to trust the index file entirely (using the *--cached* flag) or ask the diff logic to show any files that don't match the stat state as being "tentatively changed". Both of these operations are very useful indeed.

CACHED MODE

If *--cached* is specified, it allows you to ask:

```
show me the differences between HEAD and the current index
contents (the ones I'd write using 'git write-tree')
```

For example, let's say that you have worked on your working directory, updated some files in the index and are ready to commit. You want to see exactly **what** you are going to commit, without having to write a new tree object and compare it that way, and to do that, you just do

```
git diff-index --cached HEAD
```

Example: let's say I had renamed *commit.c* to *git-commit.c*, and I had done an *update-index* to make that effective in the index file. *git diff-files* wouldn't show anything at all, since the index file matches my working directory. But doing a *git diff-index* does:

```
torvalds@ppc970:~/git> git diff-index --cached HEAD
:100644 000000 4161aecc6700a2eb579e842af0b7f22b98443f74
 0000000000000000000000000000000000000000 D    commit.c
:000000 100644 0000000000000000000000000000000000000000
 4161aecc6700a2eb579e842af0b7f22b98443f74 A    git-commit.c
```

You can see easily that the above is a rename.

In fact, *git diff-index --cached* **should** always be entirely equivalent to actually doing a *git write-tree* and comparing that. Except this one is much nicer for the case where you just want to check where you are.

So doing a *git diff-index --cached* is basically very useful when you are asking yourself "what have I already marked for being committed, and what's the difference to a previous tree".

NON-CACHED MODE

The "non-cached" mode takes a different approach, and is potentially the more useful of the two in that what it does can't be emulated with a *git write-tree* + *git diff-tree*. Thus that's the default mode. The non-cached version asks the question:

```
show me the differences between HEAD and the currently checked out
tree - index contents _and_ files that aren't up to date
```

which is obviously a very useful question too, since that tells you what you **could** commit. Again, the output matches the *git diff-tree -r* output to a tee, but with a twist.

The twist is that if some file doesn't match the index, we don't have a backing store thing for it, and we use the magic "all-zero" sha1 to show that. So let's say that you have edited *kernel/sched.c*, but have not actually done a *git update-index* on it yet - there is no "object" associated with the new state, and you get:

```
torvalds@ppc970:~/v2.6/linux> git diff-index --abbrev HEAD
:100644 100644 7476bb5ba 000000000 M  kernel/sched.c
```

i.e., it shows that the tree has changed, and that *kernel/sched.c* is not up to date and may contain new stuff. The all-zero sha1 means that to get the real diff, you need to look at the object in the working directory directly rather than do an object-to-object diff.

> **Note**
>
> As with other commands of this type, *git diff-index* does not actually look at the contents of the file at all. So maybe *kernel/sched.c* hasn't actually changed, and it's just that you touched it. In either case, it's a note that you need to *git update-index* it to make the index be in sync.

> **Note**
>
> You can have a mixture of files show up as "has been updated" and "is still dirty in the working directory" together. You can always tell which file is in which state, since the "has been updated" ones show a valid sha1, and the "not in sync with the index" ones will always have the special all-zero sha1.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.43. git-diff-tree(1)

2024-07-02

Revision History

2024-07-02

NAME

git-diff-tree - Compares the content and mode of blobs found via two tree objects

SYNOPSIS

git diff-tree [--stdin] [-m] [-s] [-v] [--no-commit-id] [--pretty]
[-t] [-r] [-c | --cc] [--combined-all-paths] [--root] [--merge-base]
[<common-diff-options>] <tree-ish> [<tree-ish>] [<path>…]

DESCRIPTION

Compare the content and mode of blobs found via two tree objects.

If there is only one <tree-ish> given, the commit is compared with its parents (see --stdin below).

Note that *git diff-tree* can use the tree encapsulated in a commit object.

OPTIONS

-p , -u , --patch

Generate patch (see the section called “Generating patch text with -p”).

-s , --no-patch

Suppress all output from the diff machinery. Useful for commands like *git show* that show the patch by default to squelch their output, or to cancel the effect of options like *--patch*, *--stat* earlier on the command line in an alias.

-U<n> , --unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies *--patch*.

--output=<file>

Output to a specific file instead of stdout.

--output-indicator-new=<char> , --output-indicator-old=<char> , --output-indicator-context=<char>

Specify the character used to indicate new, old or context lines in the generated patch. Normally they are +, - and ' ' respectively.

--raw

Generate the diff in raw format. This is the default.

--patch-with-raw

Synonym for *-p --raw*.

--indent-heuristic

Enable the heuristic that shifts diff hunk boundaries to make patches easier to read. This is the default.

--no-indent-heuristic

Disable the indent heuristic.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--anchored=<text>

Generate a diff using the "anchored diff" algorithm.

This option may be specified more than once.

If a line exists in both the source and destination, exists only once, and starts with this text, this algorithm attempts to prevent it from appearing as a deletion or addition in the output. It uses the "patience diff" algorithm internally.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

default, myers

The basic greedy diff algorithm. Currently, this is the default.

minimal

Spend extra time to make sure the smallest possible diff is produced.

patience

Use "patience diff" algorithm when generating patches.

histogram

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured the *diff.algorithm* variable to a non-default value and want to use the default one, then you have to use *--diff-algorithm=default* option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by *<width>*. The width of the filename part can be limited by giving another width *<name-width>* after a comma or by setting *diff.statNameWidth=<width>*. The width of the graph part can be limited by using *--stat-graph-width=<width>* or by setting *diff.statGraphWidth=<width>*. Using *--stat* or *--stat-graph-width* affects all commands generating a stat graph, while setting *diff.statNameWidth* or *diff.statGraphWidth* does not affect *git format-patch*. By giving a third parameter *<count>*, you can limit the output to the first *<count>* lines, followed by ... if there are more.

These parameters can also be set individually with *--stat-width=<width>*, *--stat-name-width=<name-width>* and *--stat-count=<count>*.

--compact-summary

Output a condensed summary of extended header information such as file creations or deletions ("new" or "gone", optionally "+l" if it's a symlink) and mode changes ("+x" or "-x" for adding or removing executable bit respectively) in diffstat. The information is put between the filename part and the graph part. Implies *--stat*.

--numstat

Similar to *--stat*, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two - instead of saying *0 0*.

--shortstat

Output only the last line of the *--stat* format containing total number of modified files, as well as number of added and deleted lines.

-X[<param1,param2,…>] , --dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of *--dirstat* can be customized by passing it a comma separated list of parameters. The defaults are controlled by the *diff.dirstat* configuration variable (see Section G.3.29, “git-config(1)”). The following parameters are available:

changes

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

lines

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive *--dirstat* behavior than the *changes* behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other *--*stat* options.

files

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest *--dirstat* behavior, since it does not have to look at the file contents at all.

cumulative

Count changes in a child directory for the parent directory as well. Note that when using *cumulative*, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the *noncumulative* parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: *--dirstat=files,10,cumulative*.

--cumulative

Synonym for --dirstat=cumulative

--dirstat-by-file[=<param1,param2>…]

Synonym for --dirstat=files,<param1>,<param2>…

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for *-p --stat*.

-z

When *--raw*, *--numstat*, *--name-only* or *--name-status* has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

--name-only

Show only the name of each changed file in the post-image tree. The file names are often encoded in UTF-8. For more information see the discussion about encoding in the Section G.3.74, “git-log(1)” manual page.

--name-status

Show only the name(s) and status of each changed file. See the description of the *--diff-filter* option on what the status letters mean. Just like *--name-only* the file names are often encoded in UTF-8.

--submodule[=<format>]

Specify how differences in submodules are shown. When specifying *--submodule=short* the *short* format is used. This format just shows the names of the commits at the beginning and end of the range. When *--submodule* or *--submodule=log* is specified, the *log* format is used. This format lists the commits in the range like Section G.3.141, "git-submodule(1)" *summary* does. When *--submodule=diff* is specified, the *diff* format is used. This format shows an inline diff of the changes in the submodule contents between the commit range. Defaults to *diff.submodule* or the *short* format if the config option is unset.

--color[=<when>]

Show colored diff. *--color* (i.e. without =<*when*>) is the same as *--color=always*. <*when*> can be one of *always*, *never*, or *auto*.

--no-color

Turn off colored diff. It is the same as *--color=never*.

--color-moved[=<mode>]

Moved lines of code are colored differently. The <mode> defaults to *no* if the option is not given and to *zebra* if the option with no mode is given. The mode must be one of:

no

Moved lines are not highlighted.

default

Is a synonym for *zebra*. This may change to a more sensible mode in the future.

plain

Any line that is added in one location and was removed in another location will be colored with *color.diff.newMoved*. Similarly *color.diff.oldMoved* will be used for removed lines that are added somewhere else in the diff. This mode picks up any moved line, but it is not very useful in a review to determine if a block of code was moved without permutation.

blocks

Blocks of moved text of at least 20 alphanumeric characters are detected greedily. The detected blocks are painted using either the *color.diff.{old,new}Moved* color. Adjacent blocks cannot be told apart.

zebra

Blocks of moved text are detected as in *blocks* mode. The blocks are painted using either the *color.diff.{old,new}Moved* color or *color.diff.{old,new}MovedAlternative*. The change between the two colors indicates that a new block was detected.

dimmed-zebra

Similar to *zebra*, but additional dimming of uninteresting parts of moved code is performed. The bordering lines of two adjacent blocks are considered interesting, the rest is uninteresting. *dimmed_zebra* is a deprecated synonym.

--no-color-moved

Turn off move detection. This can be used to override configuration settings. It is the same as *--color-moved=no*.

--color-moved-ws=<modes>

This configures how whitespace is ignored when performing the move detection for *--color-moved*. These modes can be given as a comma separated list:

no

Do not ignore whitespace when performing move detection.

ignore-space-at-eol

Ignore changes in whitespace at EOL.

ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

allow-indentation-change

Initially ignore any whitespace in the move detection, then group the moved code blocks only into a block if the change in whitespace is the same per line. This is incompatible with the other modes.

--no-color-moved-ws

Do not ignore whitespace when performing move detection. This can be used to override configuration settings. It is the same as *--color-moved-ws=no*.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see *--word-diff-regex* below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies *--color*.

plain

Show words as *[-removed-]* and *{+added+}*. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a +/-/` ` character at the beginning of the line and extending to the end of the line. Newlines in the input are represented by a tilde ~ on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies *--word-diff* unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append *|[^[:space:]]* to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, *--word-diff-regex=.* will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see Section G.4.2, "gitattributes(5)" or Section G.3.29, "git-config(1)". Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to *--word-diff=color* plus (if a regex was specified) *--word-diff-regex=<regex>*.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--[no-]rename-empty

Whether to use empty blobs as rename source.

--check

Warn if changes introduce conflict markers or whitespace errors. What are considered whitespace errors is controlled by *core.whitespace* configuration. By default, trailing whitespaces (including lines that consist solely of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors in the *context*, *old* or *new* lines of the diff. Multiple values are separated by comma, *none* resets previous values, *default* reset the list to *new* and *all* is a shorthand for *old,new,context*. When this option is not given, and the configuration variable *diff.wsErrorHighlight* is not set, only whitespace errors in *new* lines are highlighted. The whitespace errors are colored with *color.diff.whitespace*.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to *--full-index*, output a binary diff that can be applied with *git-apply*. Implies *--patch*.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show the shortest prefix that is at least *<n>* hexdigits long that uniquely refers the object. In diff-patch output format, *--full-index* takes higher precedence, i.e. if *--full-index* is specified, full blob names will be shown regardless of *--abbrev*. Non default number of digits can be specified with *--abbrev=<n>*.

-B[<n>][/<m>] , --break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion

of everything old followed by a single insertion of everything new, and the number *m* controls this aspect of the -B option (defaults to 60%). *-B/70%* specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number *n* controls this aspect of the -B option (defaults to 50%). *-B20%* specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>] , --find-renames[=<n>]

Detect renames. If *n* is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, *-M90%* means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a *%* sign, the number is to be read as a fraction, with a decimal point before it. I.e., *-M5* becomes 0.5, and is thus the same as *-M50%*. Similarly, *-M05* is the same as *-M5%*. To limit detection to exact renames, use *-M100%*. The default similarity index is 50%.

-C[<n>] , --find-copies[=<n>]

Detect copies as well as renames. See also *--find-copies-harder*. If *n* is specified, it has the same meaning as for *-M<n>*.

--find-copies-harder

For performance reasons, by default, *-C* option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one *-C* option has the same effect.

-D , --irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and */dev/null*. The resulting patch is not meant to be applied with *patch* or *git apply*; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lacks enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with *-B*, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The *-M* and *-C* options involve some preliminary steps that can detect subsets of renames/copies cheaply, followed by an exhaustive fallback portion that compares all remaining unpaired destinations to all relevant sources. (For renames, only remaining unpaired sources are relevant; for copies, all original sources are relevant.) For N sources and destinations, this exhaustive check is O(N^2). This option prevents the exhaustive portion of rename/copy detection from running if the number of source/destination files involved exceeds the specified number. Defaults to diff.renameLimit. Note that a value of 0 is treated as unlimited.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (*A*), Copied (*C*), Deleted (*D*), Modified (*M*), Renamed (*R*), have their type (i.e. regular file, symlink, submodule, …) changed (*T*), are Unmerged (*U*), are Unknown (*X*), or have had their pairing Broken (*B*). Any combination of the filter characters (including none) can be used. When * (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

Also, these upper-case letters can be downcased to exclude. E.g. *--diff-filter=ad* excludes added and deleted paths.

Note that not all diffs can feature all types. For instance, copied and renamed entries cannot appear if detection for those types is disabled.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter's use.

It is useful when you're looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into *-S*, and keep going until you get the very first version of the block.

Binary files are searched as well.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between *-S<regex> --pickaxe-regex* and *-G<regex>*, consider a commit with the following diff in the same file:

```
+    return frotz(nitfol, two->ptr, 1, 0);
...
-    hit = frotz(nitfol, mf2.ptr, 1, 0);
```

While *git log -G"frotz\(nitfol"* will show this commit, *git log -S"frotz\(nitfol" --pickaxe-regex* will not (because the number of occurrences of that string did not change).

Unless *--text* is supplied patches of binary files without a textconv filter will be ignored.

See the *pickaxe* entry in Section G.4.4, “gitdiffcore(7)” for more information.

--find-object=<object-id>

Look for differences that change the number of occurrences of the specified object. Similar to *-S*, just the argument is different in that it doesn't search for a specific string but for a specific object id.

The object can be a blob or a submodule commit. It implies the *-t* option in *git-log* to also find trees.

--pickaxe-all

When *-S* or *-G* finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to *-S* as an extended POSIX regular expression to match.

-O<orderfile>

Control the order in which files appear in the output. This overrides the *diff.orderFile* configuration variable (see Section G.3.29, “git-config(1)”). To cancel *diff.orderFile*, use *-O/dev/null*.

The output order is determined by the order of glob patterns in <orderfile>. All files with pathnames that match the first pattern are output first, all files with pathnames that match the second pattern (but not the first) are output next, and so on. All files with pathnames that do not match any pattern are output last, as if there was an implicit match-all pattern at the end of the file. If multiple pathnames have the same rank (they match the same pattern but no earlier patterns), their output order relative to each other is the normal order.

<orderfile> is parsed as follows:

- Blank lines are ignored, so they can be used as separators for readability.
- Lines starting with a hash ("#") are ignored, so they can be used for comments. Add a backslash ("\") to the beginning of the pattern if it starts with a hash.

- Each other line contains a single pattern.

Patterns have the same syntax and semantics as patterns used for fnmatch(3) without the FNM_PATHNAME flag, except a pathname also matches a pattern if removing any number of the final pathname components matches the pattern. For example, the pattern "*foo*bar*" matches "*fooasdfbar*" and "*foo/bar/baz/asdf*" but not "*foobarx*".

--skip-to=<file> , --rotate-to=<file>

Discard the files before the named <file> from the output (i.e. *skip to*), or move them to the end of the output (i.e. *rotate to*). These options were invented primarily for the use of the *git difftool* command, and may not be very useful otherwise.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>] , --no-relative

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument. *--no-relative* can be used to countermand both *diff.relative* config option and previous *--relative*.

-a , --text

Treat all files as text.

--ignore-cr-at-eol

Ignore carriage-return at the end of line when doing a comparison.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b , --ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w , --ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

-I<regex> , --ignore-matching-lines=<regex>

Ignore changes whose all lines match <regex>. This option may be specified more than once.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other. Defaults to *diff.interHunkContext* or 0 if the config option is unset.

-W , --function-context

Show whole function as context lines for each change. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

--exit-code

Make the program exit with codes similar to diff(1). That is, it exits with 1 if there were differences and 0 means no differences.

--quiet

Disable all output of the program. Implies *--exit-code*.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with Section G.4.2, “gitattributes(5)”, you need to use this option with Section G.3.74, “git-log(1)” and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv , --no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See Section G.4.2, “gitattributes(5)” for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for Section G.3.44, “git-diff(1)” and Section G.3.74, “git-log(1)”, but not for Section G.3.54, “git-format-patch(1)” or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in Section G.3.29, “git-config(1)” or Section G.4.10, “gitmodules(5)”. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

--default-prefix

Use the default source and destination prefixes ("a/" and "b/"). This overrides configuration variables such as *diff.noprefix*, *diff.srcPrefix*, *diff.dstPrefix*, and *diff.mnemonicPrefix* (see *git-config*(1)).

--line-prefix=<prefix>

Prepend an additional prefix to every line of output.

--ita-invisible-in-index

By default entries added by "git add -N" appear as an existing empty file in "git diff" and a new file in "git diff --cached". This option makes the entry appear as a new file in "git diff" and non-existent in "git diff --

cached". This option could be reverted with *--ita-visible-in-index*. Both options are experimental and could be removed in future.

For more detailed explanation on these common options, see also Section G.4.4, “gitdiffcore(7)”.

<tree-ish>

The id of a tree object.

<path>…

If provided, the results are limited to a subset of files matching one of the provided pathspecs.

-r

Recurse into sub-trees.

-t

Show tree entry itself as well as subtrees. Implies -r.

--root

When *--root* is specified the initial commit will be shown as a big creation event. This is equivalent to a diff against the NULL tree.

--merge-base

Instead of comparing the <tree-ish>s directly, use the merge base between the two <tree-ish>s as the "before" side. There must be two <tree-ish>s given and they must both be commits.

--stdin

When *--stdin* is specified, the command does not take <tree-ish> arguments from the command line. Instead, it reads lines containing either two <tree>, one <commit>, or a list of <commit> from its standard input. (Use a single space as separator.)

When two trees are given, it compares the first tree with the second. When a single commit is given, it compares the commit with its parents. The remaining commits, when given, are used as if they are parents of the first commit.

When comparing two trees, the ID of both trees (separated by a space and terminated by a newline) is printed before the difference. When comparing commits, the ID of the first (or only) commit, followed by a newline, is printed.

The following flags further affect the behavior when comparing commits (but not trees).

-m

By default, *git diff-tree --stdin* does not show differences for merge commits. With this flag, it shows differences to that commit from all of its parents. See also *-c*.

-s

By default, *git diff-tree --stdin* shows differences, either in machine-readable form (without *-p*) or in patch form (with *-p*). This output can be suppressed. It is only useful with the *-v* flag.

-v

This flag causes *git diff-tree --stdin* to also show the commit message before the differences.

--pretty[=<format>] , --format=<format>

Pretty-print the contents of the commit logs in a given format, where *<format>* can be one of *oneline*, *short*, *medium*, *full*, *fuller*, *reference*, *email*, *raw*, *format:<string>* and *tformat:<string>*. When *<format>* is none of the above, and has *%placeholder* in it, it acts as if *--pretty=tformat:<format>* were given.

See the "PRETTY FORMATS" section for some additional details for each format. When *=<format>* part is omitted, it defaults to *medium*.

Note: you can specify the default pretty format in the repository configuration (see Section G.3.29, "git-config(1)").

--abbrev-commit

Instead of showing the full 40-byte hexadecimal commit object name, show a prefix that names the object uniquely. "--abbrev=<n>" (which also modifies diff output, if it is displayed) option can be used to specify the minimum length of the prefix.

This should make "--pretty=oneline" a whole lot more readable for people using 80-column terminals.

--no-abbrev-commit

Show the full 40-byte hexadecimal commit object name. This negates *--abbrev-commit*, either explicit or implied by other options such as "--oneline". It also overrides the *log.abbrevCommit* variable.

--oneline

This is a shorthand for "--pretty=oneline --abbrev-commit" used together.

--encoding=<encoding>

Commit objects record the character encoding used for the log message in their encoding header; this option can be used to tell the command to re-code the commit log message in the encoding preferred by the user. For non plumbing commands this defaults to UTF-8. Note that if an object claims to be encoded in *X* and we are outputting in *X*, we will output the object verbatim; this means that invalid sequences in the original commit may be copied to the output. Likewise, if iconv(3) fails to convert the commit, we will quietly output the original object verbatim.

--expand-tabs=<n> , --expand-tabs , --no-expand-tabs

Perform a tab expansion (replace each tab with enough spaces to fill to the next display column that is a multiple of *<n>*) in the log message before showing it in the output. *--expand-tabs* is a short-hand for *--expand-tabs=8*, and *--no-expand-tabs* is a short-hand for *--expand-tabs=0*, which disables tab expansion.

By default, tabs are expanded in pretty formats that indent the log message by 4 spaces (i.e. *medium*, which is the default, *full*, and *fuller*).

--notes[=<ref>]

Show the notes (see Section G.3.94, "git-notes(1)") that annotate the commit, when showing the commit log message. This is the default for *git log*, *git show* and *git whatchanged* commands when there is no *--pretty*, *--format*, or *--oneline* option given on the command line.

By default, the notes shown are from the notes refs listed in the *core.notesRef* and *notes.displayRef* variables (or corresponding environment overrides). See Section G.3.29, "git-config(1)" for more details.

With an optional *<ref>* argument, use the ref to find the notes to display. The ref can specify the full refname when it begins with *refs/notes/*; when it begins with *notes/*, *refs/* and otherwise *refs/notes/* is prefixed to form the full name of the ref.

Multiple --notes options can be combined to control which notes are being displayed. Examples: "--notes=foo" will show only notes from "refs/notes/foo"; "--notes=foo --notes" will show both notes from "refs/notes/foo" and from the default notes ref(s).

--no-notes

Do not show notes. This negates the above *--notes* option, by resetting the list of notes refs from which notes are shown. Options are parsed in the order given on the command line, so e.g. "--notes --notes=foo --no-notes --notes=bar" will only show notes from "refs/notes/bar".

--show-notes-by-default

Show the default notes unless options for displaying specific notes are given.

--show-notes[=<ref>] , --[no-]standard-notes

These options are deprecated. Use the above --notes/--no-notes options instead.

--show-signature

Check the validity of a signed commit object by passing the signature to *gpg --verify* and show the output.

--no-commit-id

git diff-tree outputs a line with the commit ID when applicable. This flag suppressed the commit ID output.

-c

This flag changes the way a merge commit is displayed (which means it is useful only when the command is given one <tree-ish>, or *--stdin*). It shows the differences from each of the parents to the merge result simultaneously instead of showing pairwise diff between a parent and the result one at a time (which is what the *-m* option does). Furthermore, it lists only files which were modified from all parents.

--cc

This flag changes the way a merge commit patch is displayed, in a similar way to the *-c* option. It implies the *-c* and *-p* options and further compresses the patch output by omitting uninteresting hunks whose contents in the parents have only two variants and the merge result picks one of them without modification. When all hunks are uninteresting, the commit itself and the commit log message are not shown, just like in any other "empty diff" case.

--combined-all-paths

This flag causes combined diffs (used for merge commits) to list the name of the file from all parents. It thus only has effect when -c or --cc are specified, and is likely only useful if filename changes are detected (i.e. when either rename or copy detection have been requested).

--always

Show the commit itself and the commit log message even if the diff itself is empty.

PRETTY FORMATS

If the commit is a merge, and if the pretty-format is not *oneline*, *email* or *raw*, an additional line is inserted before the *Author:* line. This line begins with "Merge: " and the hashes of ancestral commits are printed, separated by spaces. Note that the listed commits may not necessarily be the list of the **direct** parent commits if you have limited your view of history: for example, if you are only interested in changes related to a certain directory or file.

There are several built-in formats, and you can define additional formats by setting a pretty.<name> config option to either another format name, or a *format:* string, as described below (see Section G.3.29, “git-config(1)”). Here are the details of the built-in formats:

- *oneline*

  ```
  <hash> <title-line>
  ```

  This is designed to be as compact as possible.

- *short*

```
commit <hash>
Author: <author>

<title-line>
```

- *medium*

```
commit <hash>
Author: <author>
Date:   <author-date>

<title-line>

<full-commit-message>
```

- *full*

```
commit <hash>
Author: <author>
Commit: <committer>

<title-line>

<full-commit-message>
```

- *fuller*

```
commit <hash>
Author:     <author>
AuthorDate: <author-date>
Commit:     <committer>
CommitDate: <committer-date>

<title-line>

<full-commit-message>
```

- *reference*

```
<abbrev-hash> (<title-line>, <short-author-date>)
```

This format is used to refer to another commit in a commit message and is the same as *--pretty='format:%C(auto)%h (%s, %ad)'*. By default, the date is formatted with *--date=short* unless another *--date* option is explicitly specified. As with any *format:* with format placeholders, its output is not affected by other options like *--decorate* and *--walk-reflogs*.

- *email*

```
From <hash> <date>
From: <author>
Date: <author-date>
Subject: [PATCH] <title-line>

<full-commit-message>
```

- *mboxrd*

Like *email*, but lines in the commit message starting with "From " (preceded by zero or more ">") are quoted with ">" so they aren't confused as starting a new commit.

- *raw*

The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the hashes are displayed in full, regardless of whether --abbrev or --no-abbrev are used, and *parents* information show the true parent commits, without taking grafts or history simplification into account. Note that this format affects the way commits are displayed, but not the way the diff is shown e.g. with *git log --raw*. To get full object names in a raw diff format, use *--no-abbrev*.

- *format:<format-string>*

  The *format:<format-string>* format allows you to specify which information you want to show. It works a little bit like printf format, with the notable exception that you get a newline with *%n* instead of *\n*.

  E.g, *format:"The author of %h was %an, %ar%nThe title was >>%s<<%n"* would show something like this:

```
The author of fe6e0ee was Junio C Hamano, 23 hours ago
The title was >>t4119: test autocomputing -p<n> for traditional diff
 input.<<
```

  The placeholders are:

  - Placeholders that expand to a single literal character:

    %n

    newline

    %%

    a raw *%*

    %x00

    %x followed by two hexadecimal digits is replaced with a byte with the hexadecimal digits' value (we will call this "literal formatting code" in the rest of this document).

  - Placeholders that affect formatting of later placeholders:

    %Cred

    switch color to red

    %Cgreen

    switch color to green

    %Cblue

    switch color to blue

    %Creset

    reset color

    %C(…)

    color specification, as described under Values in the "CONFIGURATION FILE" section of Section G.3.29, “git-config(1)”. By default, colors are shown only when enabled for log output (by *color.diff*, *color.ui*, or *--color*, and respecting the *auto* settings of the former if we are going to a terminal). *%C(auto,...)* is accepted as a historical synonym for the default (e.g., *%C(auto,red)*). Specifying *%C(always,...)* will show the colors even when color is not otherwise enabled (though consider just using *--color=always* to enable color for the whole output, including this format and anything else git might color). *auto* alone (i.e. *%C(auto)*) will turn on auto coloring on the next placeholders until the color is switched again.

%m

left (<), right (>) or boundary (-) mark

%w([<w>[,<i1>[,<i2>]]])

switch line wrapping, like the -w option of Section G.3.130, “git-shortlog(1)”.

%<(<N> [,trunc|ltrunc|mtrunc])

make the next placeholder take at least N column widths, padding spaces on the right if necessary. Optionally truncate (with ellipsis ..) at the left (ltrunc) *..ft*, the middle (mtrunc) *mi..le*, or the end (trunc) *rig..*, if the output is longer than N columns. Note 1: that truncating only works correctly with N >= 2. Note 2: spaces around the N and M (see below) values are optional. Note 3: Emojis and other wide characters will take two display columns, which may over-run column boundaries. Note 4: decomposed character combining marks may be misplaced at padding boundaries.

%<|(<M>)

make the next placeholder take at least until Mth display column, padding spaces on the right if necessary. Use negative M values for column positions measured from the right hand edge of the terminal window.

%>(<N>), %>|(<M>)

similar to *%<(<N>), %<|(<M>)* respectively, but padding spaces on the left

%>>(<N>), %>>|(<M>)

similar to *%>(<N>), %>|(<M>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces

%><(<N>), %><|(<M>)

similar to *%<(<N>), %<|(<M>)* respectively, but padding both sides (i.e. the text is centered)

- Placeholders that expand to information extracted from the commit:

%H

commit hash

%h

abbreviated commit hash

%T

tree hash

%t

abbreviated tree hash

%P

parent hashes

%p

abbreviated parent hashes

%an

author name

%aN

author name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ae

author email

%aE

author email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%al

author email local-part (the part before the @ sign)

%aL

author local-part (see *%al*) respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ad

author date (format respects --date= option)

%aD

author date, RFC2822 style

%ar

author date, relative

%at

author date, UNIX timestamp

%ai

author date, ISO 8601-like format

%aI

author date, strict ISO 8601 format

%as

author date, short format (*YYYY-MM-DD*)

%ah

author date, human style (like the *--date=human* option of Section G.3.120, “git-rev-list(1)”)

%cn

committer name

%cN

committer name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ce

committer email

%cE

committer email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%cl

committer email local-part (the part before the @ sign)

%cL

committer local-part (see *%cl*) respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%cd

committer date (format respects --date= option)

%cD

committer date, RFC2822 style

%cr

committer date, relative

%ct

committer date, UNIX timestamp

%ci

committer date, ISO 8601-like format

%cI

committer date, strict ISO 8601 format

%cs

committer date, short format (*YYYY-MM-DD*)

%ch

committer date, human style (like the *--date=human* option of Section G.3.120, “git-rev-list(1)”)

%d

ref names, like the --decorate option of Section G.3.74, “git-log(1)”

%D

ref names without the " (", ")" wrapping.

%(decorate[:<options>])

ref names with custom decorations. The *decorate* string may be followed by a colon and zero or more comma-separated options. Option values may contain literal formatting codes. These must be used for commas (*%x2C*) and closing parentheses (*%x29*), due to their role in the option syntax.

- *prefix=<value>*: Shown before the list of ref names. Defaults to " *(*".
- *suffix=<value>*: Shown after the list of ref names. Defaults to "*)*".
- *separator=<value>*: Shown between ref names. Defaults to ", ".
- *pointer=<value>*: Shown between HEAD and the branch it points to, if any. Defaults to " -> ".
- *tag=<value>*: Shown before tag names. Defaults to "*tag:* ".

For example, to produce decorations with no wrapping or tag annotations, and spaces as separators:

%(decorate:prefix=,suffix=,tag=,separator=)

%(describe[:<options>])

human-readable name, like Section G.3.39, “git-describe(1)”; empty string for undescribable commits. The *describe* string may be followed by a colon and zero or more comma-separated options. Descriptions can be inconsistent when tags are added or removed at the same time.

- *tags[=<bool-value>]*: Instead of only considering annotated tags, consider lightweight tags as well.
- *abbrev=<number>*: Instead of using the default number of hexadecimal digits (which will vary according to the number of objects in the repository with a default of 7) of the abbreviated object name, use <number> digits, or as many digits as needed to form a unique object name.
- *match=<pattern>*: Only consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix.
- *exclude=<pattern>*: Do not consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix.

%S

ref name given on the command line by which the commit was reached (like *git log --source*), only works with *git log*

%e

encoding

%s

subject

%f

sanitized subject line, suitable for a filename

%b

body

%B

raw body (unwrapped subject and body)

%N

commit notes

%GG

raw verification message from GPG for a signed commit

%G?

show "G" for a good (valid) signature, "B" for a bad signature, "U" for a good signature with unknown validity, "X" for a good signature that has expired, "Y" for a good signature made by an expired key, "R" for a good signature made by a revoked key, "E" if the signature cannot be checked (e.g. missing key) and "N" for no signature

%GS

show the name of the signer for a signed commit

%GK

show the key used to sign a signed commit

%GF

show the fingerprint of the key used to sign a signed commit

%GP

show the fingerprint of the primary key whose subkey was used to sign a signed commit

%GT

show the trust level for the key used to sign a signed commit

%gD

reflog selector, e.g., *refs/stash@{1}* or *refs/stash@{2 minutes ago}*; the format follows the rules described for the *-g* option. The portion before the @ is the refname as given on the command line (so *git log -g refs/heads/master* would yield *refs/heads/master@{0}*).

%gd

shortened reflog selector; same as *%gD*, but the refname portion is shortened for human readability (so *refs/heads/master* becomes just *master*).

%gn

reflog identity name

%gN

reflog identity name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ge

reflog identity email

%gE

reflog identity email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%gs

reflog subject

%(trailers[:<options>])

display the trailers of the body as interpreted by Section G.3.73, “git-interpret-trailers(1)”. The *trailers* string may be followed by a colon and zero or more comma-separated options. If any option is provided multiple times, the last occurrence wins.

- *key=<key>*: only show trailers with specified <key>. Matching is done case-insensitively and trailing colon is optional. If option is given multiple times trailer lines matching any of the keys are shown. This option automatically enables the *only* option so that non-trailer lines in the trailer block are hidden. If that is not desired it can be disabled with *only=false*. E.g., *%(trailers:key=Reviewed-by)* shows trailer lines with key *Reviewed-by*.
- *only[=<bool>]*: select whether non-trailer lines from the trailer block should be included.
- *separator=<sep>*: specify the separator inserted between trailer lines. Defaults to a line feed character. The string <sep> may contain the literal formatting codes described above. To use comma as separator one must use *%x2C* as it would otherwise be parsed as next option. E.g., *%(trailers:key=Ticket,separator=%x2C)* shows all trailer lines whose key is "Ticket" separated by a comma and a space.
- *unfold[=<bool>]*: make it behave as if interpret-trailer's *--unfold* option was given. E.g., *%(trailers:only,unfold=true)* unfolds and shows all trailer lines.
- *keyonly[=<bool>]*: only show the key part of the trailer.
- *valueonly[=<bool>]*: only show the value part of the trailer.
- *key_value_separator=<sep>*: specify the separator inserted between the key and value of each trailer. Defaults to ": ". Otherwise it shares the same semantics as *separator=<sep>* above.

> **Note**
>
> Some placeholders may depend on other options given to the revision traversal engine. For example, the *%g** reflog options will insert an empty string unless we are traversing reflog entries (e.g., by *git log -g*). The *%d* and *%D* placeholders will use the "short" decoration format if *--decorate* was not already provided on the command line.

The boolean options accept an optional value *[=<bool-value>]*. The values *true*, *false*, *on*, *off* etc. are all accepted. See the "boolean" sub-section in "EXAMPLES" in Section G.3.29, “git-config(1)”. If a boolean option is given with no value, it's enabled.

If you add a + (plus sign) after % of a placeholder, a line-feed is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

If you add a - (minus sign) after % of a placeholder, all consecutive line-feeds immediately preceding the expansion are deleted if and only if the placeholder expands to an empty string.

If you add a `` ` `` (space) after *%* of a placeholder, a space is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

- *tformat:*

  The *tformat:* format works exactly like *format:*, except that it provides "terminator" semantics instead of "separator" semantics. In other words, each commit has the message terminator character (usually a newline) appended, rather than a separator placed between entries. This means that the final entry of a single-line format will be properly terminated with a new line, just as the "oneline" format does. For example:

  ```
  $ git log -2 --pretty=format:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973 -- NO NEWLINE

  $ git log -2 --pretty=tformat:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973
  ```

  In addition, any unrecognized string that has a *%* in it is interpreted as if it has *tformat:* in front of it. For example, these two are equivalent:

  ```
  $ git log -2 --pretty=tformat:%h 4da45bef
  $ git log -2 --pretty=%h 4da45bef
  ```

Raw output format

The raw output format from "git-diff-index", "git-diff-tree", "git-diff-files" and "git diff --raw" are very similar.

These commands all compare two sets of things; what is compared differs:

git-diff-index <tree-ish>

compares the <tree-ish> and the files on the filesystem.

git-diff-index --cached <tree-ish>

compares the <tree-ish> and the index.

git-diff-tree [-r] <tree-ish-1> <tree-ish-2> [<pattern>…]

compares the trees named by the two arguments.

git-diff-files [<pattern>…]

compares the index and the files on the filesystem.

The "git-diff-tree" command begins its output by printing the hash of what is being compared. After that, all the commands print one output line per changed file.

An output line is formatted this way:

```
in-place edit  :100644 100644 bcd1234 0123456 M file0
copy-edit      :100644 100644 abcd123 1234567 C68 file1 file2
rename-edit    :100644 100644 abcd123 1234567 R86 file1 file3
create         :000000 100644 0000000 1234567 A file4
delete         :100644 000000 1234567 0000000 D file5
unmerged       :000000 000000 0000000 0000000 U file6
```

That is, from the left to the right:

1. a colon.
2. mode for "src"; 000000 if creation or unmerged.
3. a space.
4. mode for "dst"; 000000 if deletion or unmerged.
5. a space.
6. sha1 for "src"; 0{40} if creation or unmerged.
7. a space.
8. sha1 for "dst"; 0{40} if deletion, unmerged or "work tree out of sync with the index".
9. a space.
10. status, followed by optional "score" number.
11. a tab or a NUL when *-z* option is used.
12. path for "src"
13. a tab or a NUL when *-z* option is used; only exists for C or R.
14. path for "dst"; only exists for C or R.
15. an LF or a NUL when *-z* option is used, to terminate the record.

Possible status letters are:

- A: addition of a file
- C: copy of a file into a new one
- D: deletion of a file
- M: modification of the contents or mode of a file
- R: renaming of a file
- T: change in the type of the file (regular file, symbolic link or submodule)
- U: file is unmerged (you must complete the merge before it can be committed)
- X: "unknown" change type (most probably a bug, please report it)

Status letters C and R are always followed by a score (denoting the percentage of similarity between the source and target of the move or copy). Status letter M may be followed by a score (denoting the percentage of dissimilarity) for file rewrites.

The sha1 for "dst" is shown as all 0's if a file on the filesystem is out of sync with the index.

Example:

```
:100644 100644 5be4a4a 0000000 M file.c
```

Without the *-z* option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). Using *-z* the filename is output verbatim and the line is terminated by a NUL byte.

diff format for merges

"git-diff-tree", "git-diff-files" and "git-diff --raw" can take *-c* or *--cc* option to generate diff output also for merge commits. The output differs from the format described above in the following way:

1. there is a colon for each parent
2. there are more "src" modes and "src" sha1
3. status is concatenated status characters for each parent
4. no optional "score" number
5. tab-separated pathname(s) of the file

For *-c* and *--cc*, only the destination or final path is shown even if the file was renamed on any side of history. With *--combined-all-paths*, the name of the path in each parent is shown followed by the name of the path in the merge commit.

Examples for *-c* and *--cc* without *--combined-all-paths*:

```
::100644 100644 100644 fabadb8 cc95eb0 4866510 MM       desc.c
::100755 100755 100755 52b7a2d 6d1ac04 d2ac7d7 RM       bar.sh
::100644 100644 100644 e07d6c5 9042e82 ee91881 RR       phooey.c
```

Examples when *--combined-all-paths* added to either *-c* or *--cc*:

```
::100644 100644 100644 fabadb8 cc95eb0 4866510 MM       desc.c  desc.c
 desc.c
::100755 100755 100755 52b7a2d 6d1ac04 d2ac7d7 RM       foo.sh  bar.sh
 bar.sh
::100644 100644 100644 e07d6c5 9042e82 ee91881 RR       fooey.c fuey.c
 phooey.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patch text with -p

Running Section G.3.44, “git-diff(1)”, Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, Section G.3.42, “git-diff-index(1)”, Section G.3.43, “git-diff-tree(1)”, or Section G.3.41, “git-diff-files(1)” with the *-p* option produces patch text. You can customize the creation of patch text via the *GIT_EXTERNAL_DIFF* and the *GIT_DIFF_OPTS* environment variables (see Section G.3.1, “git(1)”), and the *diff* attribute (see Section G.4.2, “gitattributes(5)”).

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded by a "git diff" header that looks like this:

   ```
   diff --git a/file1 b/file2
   ```

   The *a/* and *b/* filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, */dev/null* is *not* used in place of the *a/* or *b/* filenames.

   When a rename/copy is involved, *file1* and *file2* show the name of the source file of the rename/copy and the name of the file that the rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

   ```
   old mode <mode>
   new mode <mode>
   deleted file mode <mode>
   new file mode <mode>
   ```

```
copy from <path>
copy to <path>
rename from <path>
rename to <path>
similarity index <number>
dissimilarity index <number>
index <hash>..<hash> <mode>
```

File modes are printed as 6-digit octal numbers including the file type and file permission bits.

Path names in extended headers do not include the *a/* and *b/* prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the blob object names before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. Pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

4. All the *file1* files in the output refer to files before the commit, and all the *file2* files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

5. Hunk headers mention the name of the function to which the hunk applies. See "Defining a custom hunk-header" in Section G.4.2, “gitattributes(5)” for details of how to tailor this to specific languages.

Combined diff format

Any diff-generating command can take the *-c* or *--cc* option to produce a *combined diff* when showing a merge. This is the default format when showing merges with Section G.3.44, “git-diff(1)” or Section G.3.134, “git-show(1)”. Note also that you can give suitable *--diff-merges* option to any of these commands to force generation of diffs in a specific format.

A "combined diff" format looks like this:

```
diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
        return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +      unsigned char sha1[20];
 +      struct commit *cmit;
        struct commit_list *list;
```

```
        static int initialized = 0;
        struct commit_name *n;

+       if (get_sha1(arg, sha1) < 0)
+               usage(describe_usage);
+       cmit = lookup_commit_reference(sha1);
+       if (!cmit)
+               usage(describe_usage);
+
        if (!initialized) {
                initialized = 1;
                for_each_ref(get_name);
```

1. It is preceded by a "git diff" header, that looks like this (when the *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when the *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index <hash>,<hash>..<hash>
   mode <mode>,<mode>..<mode>
   new file mode <mode>
   deleted file mode <mode>,<mode>
   ```

   The *mode <mode>,<mode>..<mode>* line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected content movement (renames and copying detection) are designed to work with the diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by a two-line from-file/to-file header:

   ```
   --- a/file
   +++ b/file
   ```

   Similar to the two-line header for the traditional *unified* diff format, */dev/null* is used to signal created or deleted files.

   However, if the --combined-all-paths option is provided, instead of a two-line from-file/to-file, you get an N +1 line from-file/to-file header, where N is the number of parents in the merge commit:

   ```
   --- a/file
   --- a/file
   --- a/file
   +++ b/file
   ```

   This extended format can be useful if rename or copy detection is active, to allow you to see the original name of the file in different parents.

4. Chunk header format is modified to prevent people from accidentally feeding it to *patch -p1*. Combined diff format was created for review of merge commit changes, and was not meant to be applied. The change is similar to the change in the extended *index* header:

   ```
   @@@ <from-file-range> <from-file-range> <to-file-range> @@@
   ```

   There are (number of parents + 1) @ characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has - (minus -- appears in A but removed in B), + (plus -- missing in A but added to B), or " " (space -- unchanged) prefix, this

format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X's line is different from it.

A - character in the column N means that the line appears in fileN but it does not appear in the result. A + character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two - removals from both file1 and file2, plus ++ to mean one line that was added does not appear in either file1 or file2). Also, eight other lines are the same from file1 but do not appear in file2 (hence prefixed with +).

When shown by *git diff-tree -c*, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by *git diff-files -c*, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

other diff formats

The *--summary* option describes newly added, deleted, renamed and copied files. The *--stat* option adds diffstat(1) graph to the output. These options can be combined with other options, such as *-p*, and are meant for human consumption.

When showing a change that involves a rename or a copy, *--stat* output formats the pathnames compactly by combining common prefix and suffix of the pathnames. For example, a change that moves *arch/i386/Makefile* to *arch/x86/Makefile* while modifying 4 lines will be shown like this:

```
arch/{i386 => x86}/Makefile    |   4 +--
```

The *--numstat* option gives the diffstat(1) information but is designed for easier machine consumption. An entry in *--numstat* output looks like this:

```
1       2       README
3       1       arch/{i386 => x86}/Makefile
```

That is, from left to right:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. pathname (possibly with rename/copy information);
6. a newline.

When *-z* output option is in effect, the output is formatted this way:

```
1       2       README NUL
3       1       NUL arch/i386/Makefile NUL arch/x86/Makefile NUL
```

That is:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;

5. a NUL (only exists if renamed/copied);
6. pathname in preimage;
7. a NUL (only exists if renamed/copied);
8. pathname in postimage (only exists if renamed/copied);
9. a NUL.

The extra *NUL* before the preimage path in renamed case is to allow scripts that read the output to tell if the current record being read is a single-path record or a rename/copy record without reading ahead. After reading added and deleted lines, reading up to *NUL* would yield the pathname, but if that is *NUL*, the record will show two paths.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.44. git-diff(1)

2024-07-02
Revision History
2024-07-02

NAME

git-diff - Show changes between commits, commit and working tree, etc

SYNOPSIS

git diff [<options>] [<commit>] [--] [<path>…]
git diff [<options>] --cached [--merge-base] [<commit>] [--] [<path>…]
git diff [<options>] [--merge-base] <commit> [<commit>…] <commit> [--] [<path>…]
git diff [<options>] <commit>…<commit> [--] [<path>…]
git diff [<options>] <blob> <blob>
git diff [<options>] --no-index [--] <path> <path>

DESCRIPTION

Show changes between the working tree and the index or a tree, changes between the index and a tree, changes between two trees, changes resulting from a merge, changes between two blob objects, or changes between two files on disk.

git diff [<options>] [--] [<path>…]

This form is to view the changes you made relative to the index (staging area for the next commit). In other words, the differences are what you *could* tell Git to further add to the index but you still haven't. You can stage these changes by using Section G.3.2, “git-add(1)”.

git diff [<options>] --no-index [--] <path> <path>

This form is to compare the given two paths on the filesystem. You can omit the *--no-index* option when running the command in a working tree controlled by Git and at least one of the paths points outside the working tree, or when running the command outside a working tree controlled by Git. This form implies *--exit-code*.

git diff [<options>] --cached [--merge-base] [<commit>] [--] [<path>…]

This form is to view the changes you staged for the next commit relative to the named <commit>. Typically you would want comparison with the latest commit, so if you do not give <commit>, it defaults to HEAD. If HEAD does not exist (e.g. unborn branches) and <commit> is not given, it shows all staged changes. --staged is a synonym of --cached.

If --merge-base is given, instead of using <commit>, use the merge base of <commit> and HEAD. *git diff --cached --merge-base A* is equivalent to *git diff --cached $(git merge-base A HEAD)*.

git diff [<options>] [--merge-base] <commit> [--] [<path>…]

This form is to view the changes you have in your working tree relative to the named <commit>. You can use HEAD to compare it with the latest commit, or a branch name to compare with the tip of a different branch.

If --merge-base is given, instead of using <commit>, use the merge base of <commit> and HEAD. *git diff --merge-base A* is equivalent to *git diff $(git merge-base A HEAD)*.

git diff [<options>] [--merge-base] <commit> <commit> [--] [<path>…]

This is to view the changes between two arbitrary <commit>.

If --merge-base is given, use the merge base of the two commits for the "before" side. *git diff --merge-base A B* is equivalent to *git diff $(git merge-base A B) B*.

git diff [<options>] <commit> <commit>… <commit> [--] [<path>…]

This form is to view the results of a merge commit. The first listed <commit> must be the merge itself; the remaining two or more commits should be its parents. Convenient ways to produce the desired set of revisions are to use the suffixes ^@ and ^!. If A is a merge commit, then *git diff A A^@*, *git diff A^!* and *git show A* all give the same combined diff.

git diff [<options>] <commit>..<commit> [--] [<path>…]

This is synonymous to the earlier form (without the ..) for viewing the changes between two arbitrary <commit>. If <commit> on one side is omitted, it will have the same effect as using HEAD instead.

git diff [<options>] <commit>...<commit> [--] [<path>…]

This form is to view the changes on the branch containing and up to the second <commit>, starting at a common ancestor of both <commit>. *git diff A...B* is equivalent to *git diff $(git merge-base A B) B*. You can omit any one of <commit>, which has the same effect as using HEAD instead.

Just in case you are doing something exotic, it should be noted that all of the <commit> in the above description, except in the *--merge-base* case and in the last two forms that use .. notations, can be any <tree>. A tree of interest is the one pointed to by the ref named *AUTO_MERGE*, which is written by the *ort* merge strategy upon hitting merge conflicts (see Section G.3.86, “git-merge(1)”). Comparing the working tree with *AUTO_MERGE* shows changes you've made so far to resolve textual conflicts (see the examples below).

For a more complete list of ways to spell <commit>, see "SPECIFYING REVISIONS" section in Section G.4.14, “gitrevisions(7)”. However, "diff" is about comparing two *endpoints*, not ranges, and the range notations (*<commit>..<commit>* and *<commit>...<commit>*) do not mean a range as defined in the "SPECIFYING RANGES" section in Section G.4.14, “gitrevisions(7)”.

git diff [<options>] <blob> <blob>

This form is to view the differences between the raw contents of two blob objects.

OPTIONS

-p , -u , --patch

Generate patch (see the section called “Generating patch text with -p”). This is the default.

-s , --no-patch

Suppress all output from the diff machinery. Useful for commands like *git show* that show the patch by default to squelch their output, or to cancel the effect of options like *--patch*, *--stat* earlier on the command line in an alias.

-U<n> , --unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies *--patch*.

--output=<file>

Output to a specific file instead of stdout.

--output-indicator-new=<char> , --output-indicator-old=<char> , --output-indicator-context=<char>

Specify the character used to indicate new, old or context lines in the generated patch. Normally they are +, - and ' ' respectively.

--raw

Generate the diff in raw format.

--patch-with-raw

Synonym for *-p --raw*.

--indent-heuristic

Enable the heuristic that shifts diff hunk boundaries to make patches easier to read. This is the default.

--no-indent-heuristic

Disable the indent heuristic.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--anchored=<text>

Generate a diff using the "anchored diff" algorithm.

This option may be specified more than once.

If a line exists in both the source and destination, exists only once, and starts with this text, this algorithm attempts to prevent it from appearing as a deletion or addition in the output. It uses the "patience diff" algorithm internally.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

default, myers

The basic greedy diff algorithm. Currently, this is the default.

minimal

Spend extra time to make sure the smallest possible diff is produced.

patience

Use "patience diff" algorithm when generating patches.

histogram

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured the *diff.algorithm* variable to a non-default value and want to use the default one, then you have to use *--diff-algorithm=default* option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by *<width>*. The width of the filename part can be limited by giving another width *<name-width>* after a comma or by setting *diff.statNameWidth=<width>*. The width of the graph part can be limited by using *--stat-graph-width=<width>* or by setting *diff.statGraphWidth=<width>*. Using *--stat* or *--stat-graph-width* affects all commands generating a stat graph, while setting *diff.statNameWidth* or *diff.statGraphWidth* does not affect *git format-patch*. By giving a third parameter *<count>*, you can limit the output to the first *<count>* lines, followed by ... if there are more.

These parameters can also be set individually with *--stat-width=<width>*, *--stat-name-width=<name-width>* and *--stat-count=<count>*.

--compact-summary

Output a condensed summary of extended header information such as file creations or deletions ("new" or "gone", optionally "+l" if it's a symlink) and mode changes ("+x" or "-x" for adding or removing executable bit respectively) in diffstat. The information is put between the filename part and the graph part. Implies *--stat*.

--numstat

Similar to *--stat*, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two - instead of saying *0 0*.

--shortstat

Output only the last line of the *--stat* format containing total number of modified files, as well as number of added and deleted lines.

-X[<param1,param2,…>] , --dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of *--dirstat* can be customized by passing it a comma separated list of parameters. The defaults are controlled by the *diff.dirstat* configuration variable (see Section G.3.29, “git-config(1)”). The following parameters are available:

changes

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

lines

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive *--dirstat* behavior than the *changes* behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other *--*stat* options.

files

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest *--dirstat* behavior, since it does not have to look at the file contents at all.

cumulative

Count changes in a child directory for the parent directory as well. Note that when using *cumulative*, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the *noncumulative* parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: *--dirstat=files,10,cumulative*.

--cumulative

Synonym for --dirstat=cumulative

--dirstat-by-file[=<param1,param2>…]

Synonym for --dirstat=files,<param1>,<param2>…

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for *-p --stat*.

-z

When *--raw*, *--numstat*, *--name-only* or *--name-status* has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

--name-only

Show only the name of each changed file in the post-image tree. The file names are often encoded in UTF-8. For more information see the discussion about encoding in the Section G.3.74, “git-log(1)” manual page.

--name-status

Show only the name(s) and status of each changed file. See the description of the *--diff-filter* option on what the status letters mean. Just like *--name-only* the file names are often encoded in UTF-8.

--submodule[=<format>]

Specify how differences in submodules are shown. When specifying *--submodule=short* the *short* format is used. This format just shows the names of the commits at the beginning and end of the range. When *--submodule* or *--submodule=log* is specified, the *log* format is used. This format lists the commits in the range like Section G.3.141, “git-submodule(1)” *summary* does. When *--submodule=diff* is specified, the *diff* format is used. This format shows an inline diff of the changes in the submodule contents between the commit range. Defaults to *diff.submodule* or the *short* format if the config option is unset.

--color[=<when>]

Show colored diff. *--color* (i.e. without =*<when>*) is the same as *--color=always*. *<when>* can be one of *always*, *never*, or *auto*. It can be changed by the *color.ui* and *color.diff* configuration settings.

--no-color

Turn off colored diff. This can be used to override configuration settings. It is the same as *--color=never*.

--color-moved[=<mode>]

Moved lines of code are colored differently. It can be changed by the *diff.colorMoved* configuration setting. The <mode> defaults to *no* if the option is not given and to *zebra* if the option with no mode is given. The mode must be one of:

no

Moved lines are not highlighted.

default

Is a synonym for *zebra*. This may change to a more sensible mode in the future.

plain

Any line that is added in one location and was removed in another location will be colored with *color.diff.newMoved*. Similarly *color.diff.oldMoved* will be used for removed lines that are added somewhere else in the diff. This mode picks up any moved line, but it is not very useful in a review to determine if a block of code was moved without permutation.

blocks

Blocks of moved text of at least 20 alphanumeric characters are detected greedily. The detected blocks are painted using either the *color.diff.{old,new}Moved* color. Adjacent blocks cannot be told apart.

zebra

Blocks of moved text are detected as in *blocks* mode. The blocks are painted using either the *color.diff.{old,new}Moved* color or *color.diff.{old,new}MovedAlternative*. The change between the two colors indicates that a new block was detected.

dimmed-zebra

Similar to *zebra*, but additional dimming of uninteresting parts of moved code is performed. The bordering lines of two adjacent blocks are considered interesting, the rest is uninteresting. *dimmed_zebra* is a deprecated synonym.

--no-color-moved

Turn off move detection. This can be used to override configuration settings. It is the same as *--color-moved=no*.

--color-moved-ws=<modes>

This configures how whitespace is ignored when performing the move detection for *--color-moved*. It can be set by the *diff.colorMovedWS* configuration setting. These modes can be given as a comma separated list:

no

Do not ignore whitespace when performing move detection.

ignore-space-at-eol

Ignore changes in whitespace at EOL.

ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

allow-indentation-change

Initially ignore any whitespace in the move detection, then group the moved code blocks only into a block if the change in whitespace is the same per line. This is incompatible with the other modes.

--no-color-moved-ws

Do not ignore whitespace when performing move detection. This can be used to override configuration settings. It is the same as *--color-moved-ws=no*.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see *--word-diff-regex* below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies *--color*.

plain

Show words as *[-removed-]* and *{+added+}*. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a +/-/` ` character at the beginning of the line and extending to the end of the line. Newlines in the input are represented by a tilde ~ on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies *--word-diff* unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append *|[^[:space:]]* to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, *--word-diff-regex=.* will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see Section G.4.2, “gitattributes(5)” or Section G.3.29, “git-config(1)”. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to *--word-diff=color* plus (if a regex was specified) *--word-diff-regex=<regex>*.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--[no-]rename-empty

Whether to use empty blobs as rename source.

--check

Warn if changes introduce conflict markers or whitespace errors. What are considered whitespace errors is controlled by *core.whitespace* configuration. By default, trailing whitespaces (including lines that consist solely of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors in the *context*, *old* or *new* lines of the diff. Multiple values are separated by comma, *none* resets previous values, *default* reset the list to *new* and *all* is a shorthand for *old,new,context*. When this option is not given, and the configuration variable *diff.wsErrorHighlight* is not set, only whitespace errors in *new* lines are highlighted. The whitespace errors are colored with *color.diff.whitespace*.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to *--full-index*, output a binary diff that can be applied with *git-apply*. Implies *--patch*.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show the shortest prefix that is at least *<n>* hexdigits long that uniquely refers the object. In diff-patch output format, *--full-index* takes higher precedence, i.e. if *--full-index* is specified, full blob names will be shown regardless of *--abbrev*. Non default number of digits can be specified with *--abbrev=<n>*.

-B[<n>][/<m>] , --break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything new, and the number *m* controls this aspect of the -B option (defaults to 60%). *-B/70%* specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number *n* controls this aspect of the -B option (defaults to 50%). *-B20%* specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>] , --find-renames[=<n>]

Detect renames. If *n* is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, *-M90%* means Git should consider a delete/add pair to be a rename

if more than 90% of the file hasn't changed. Without a *%* sign, the number is to be read as a fraction, with a decimal point before it. I.e., *-M5* becomes 0.5, and is thus the same as *-M50%*. Similarly, *-M05* is the same as *-M5%*. To limit detection to exact renames, use *-M100%*. The default similarity index is 50%.

-C[<n>] , --find-copies[=<n>]

Detect copies as well as renames. See also *--find-copies-harder*. If *n* is specified, it has the same meaning as for *-M<n>*.

--find-copies-harder

For performance reasons, by default, *-C* option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one *-C* option has the same effect.

-D , --irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and */dev/null*. The resulting patch is not meant to be applied with *patch* or *git apply*; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lacks enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with *-B*, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The *-M* and *-C* options involve some preliminary steps that can detect subsets of renames/copies cheaply, followed by an exhaustive fallback portion that compares all remaining unpaired destinations to all relevant sources. (For renames, only remaining unpaired sources are relevant; for copies, all original sources are relevant.) For N sources and destinations, this exhaustive check is O(N^2). This option prevents the exhaustive portion of rename/copy detection from running if the number of source/destination files involved exceeds the specified number. Defaults to diff.renameLimit. Note that a value of 0 is treated as unlimited.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (*A*), Copied (*C*), Deleted (*D*), Modified (*M*), Renamed (*R*), have their type (i.e. regular file, symlink, submodule, …) changed (*T*), are Unmerged (*U*), are Unknown (*X*), or have had their pairing Broken (*B*). Any combination of the filter characters (including none) can be used. When * (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

Also, these upper-case letters can be downcased to exclude. E.g. *--diff-filter=ad* excludes added and deleted paths.

Note that not all diffs can feature all types. For instance, copied and renamed entries cannot appear if detection for those types is disabled.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter's use.

It is useful when you're looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into *-S*, and keep going until you get the very first version of the block.

Binary files are searched as well.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between *-S<regex> --pickaxe-regex* and *-G<regex>*, consider a commit with the following diff in the same file:

```
+    return frotz(nitfol, two->ptr, 1, 0);
...
-    hit = frotz(nitfol, mf2.ptr, 1, 0);
```

While *git log -G"frotz\(nitfol"* will show this commit, *git log -S"frotz\(nitfol" --pickaxe-regex* will not (because the number of occurrences of that string did not change).

Unless *--text* is supplied patches of binary files without a textconv filter will be ignored.

See the *pickaxe* entry in Section G.4.4, “gitdiffcore(7)” for more information.

--find-object=<object-id>

Look for differences that change the number of occurrences of the specified object. Similar to *-S*, just the argument is different in that it doesn't search for a specific string but for a specific object id.

The object can be a blob or a submodule commit. It implies the *-t* option in *git-log* to also find trees.

--pickaxe-all

When *-S* or *-G* finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to *-S* as an extended POSIX regular expression to match.

-O<orderfile>

Control the order in which files appear in the output. This overrides the *diff.orderFile* configuration variable (see Section G.3.29, “git-config(1)”). To cancel *diff.orderFile*, use *-O/dev/null*.

The output order is determined by the order of glob patterns in <orderfile>. All files with pathnames that match the first pattern are output first, all files with pathnames that match the second pattern (but not the first) are output next, and so on. All files with pathnames that do not match any pattern are output last, as if there was an implicit match-all pattern at the end of the file. If multiple pathnames have the same rank (they match the same pattern but no earlier patterns), their output order relative to each other is the normal order.

<orderfile> is parsed as follows:

- Blank lines are ignored, so they can be used as separators for readability.
- Lines starting with a hash ("#") are ignored, so they can be used for comments. Add a backslash ("\") to the beginning of the pattern if it starts with a hash.
- Each other line contains a single pattern.

Patterns have the same syntax and semantics as patterns used for fnmatch(3) without the FNM_PATHNAME flag, except a pathname also matches a pattern if removing any number of the final pathname components matches the pattern. For example, the pattern "*foo*bar*" matches "*fooasdfbar*" and "*foo/bar/baz/asdf*" but not "*foobarx*".

--skip-to=<file> , --rotate-to=<file>

Discard the files before the named <file> from the output (i.e. *skip to*), or move them to the end of the output (i.e. *rotate to*). These options were invented primarily for the use of the *git difftool* command, and may not be very useful otherwise.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>] , --no-relative

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument. *--no-relative* can be used to countermand both *diff.relative* config option and previous *--relative*.

-a , --text

Treat all files as text.

--ignore-cr-at-eol

Ignore carriage-return at the end of line when doing a comparison.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b , --ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w , --ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

-I<regex> , --ignore-matching-lines=<regex>

Ignore changes whose all lines match <regex>. This option may be specified more than once.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other. Defaults to *diff.interHunkContext* or 0 if the config option is unset.

-W , --function-context

Show whole function as context lines for each change. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

--exit-code

Make the program exit with codes similar to diff(1). That is, it exits with 1 if there were differences and 0 means no differences.

--quiet

Disable all output of the program. Implies *--exit-code*.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with Section G.4.2, “gitattributes(5)”, you need to use this option with Section G.3.74, “git-log(1)” and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv , --no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See Section G.4.2, “gitattributes(5)” for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for Section G.3.44, “git-diff(1)” and Section G.3.74, “git-log(1)”, but not for Section G.3.54, “git-format-patch(1)” or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in Section G.3.29, “git-config(1)” or Section G.4.10, “gitmodules(5)”. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

--default-prefix

Use the default source and destination prefixes ("a/" and "b/"). This overrides configuration variables such as *diff.noprefix*, *diff.srcPrefix*, *diff.dstPrefix*, and *diff.mnemonicPrefix* (see *git-config*(1)).

--line-prefix=<prefix>

Prepend an additional prefix to every line of output.

--ita-invisible-in-index

By default entries added by "git add -N" appear as an existing empty file in "git diff" and a new file in "git diff --cached". This option makes the entry appear as a new file in "git diff" and non-existent in "git diff --cached". This option could be reverted with *--ita-visible-in-index*. Both options are experimental and could be removed in future.

For more detailed explanation on these common options, see also Section G.4.4, “gitdiffcore(7)”.

-1 --base , -2 --ours , -3 --theirs

Compare the working tree with the "base" version (stage #1), "our branch" (stage #2) or "their branch" (stage #3). The index contains these stages only for unmerged entries i.e. while resolving conflicts. See Section G.3.106, “git-read-tree(1)” section "3-Way Merge" for detailed information.

-0

Omit diff output for unmerged entries and just show "Unmerged". Can be used only when comparing the working tree with the index.

<path>…

The <paths> parameters, when given, are used to limit the diff to the named paths (you can give directory names and get diff for all files under them).

Raw output format

The raw output format from "git-diff-index", "git-diff-tree", "git-diff-files" and "git diff --raw" are very similar.

These commands all compare two sets of things; what is compared differs:

git-diff-index <tree-ish>

compares the <tree-ish> and the files on the filesystem.

git-diff-index --cached <tree-ish>

compares the <tree-ish> and the index.

git-diff-tree [-r] <tree-ish-1> <tree-ish-2> [<pattern>…]

compares the trees named by the two arguments.

git-diff-files [<pattern>…]

compares the index and the files on the filesystem.

The "git-diff-tree" command begins its output by printing the hash of what is being compared. After that, all the commands print one output line per changed file.

An output line is formatted this way:

```
in-place edit  :100644 100644 bcd1234 0123456 M file0
copy-edit      :100644 100644 abcd123 1234567 C68 file1 file2
rename-edit    :100644 100644 abcd123 1234567 R86 file1 file3
create         :000000 100644 0000000 1234567 A file4
delete         :100644 000000 1234567 0000000 D file5
unmerged       :000000 000000 0000000 0000000 U file6
```

That is, from the left to the right:

1. a colon.
2. mode for "src"; 000000 if creation or unmerged.
3. a space.
4. mode for "dst"; 000000 if deletion or unmerged.
5. a space.
6. sha1 for "src"; 0{40} if creation or unmerged.
7. a space.
8. sha1 for "dst"; 0{40} if deletion, unmerged or "work tree out of sync with the index".
9. a space.
10. status, followed by optional "score" number.
11. a tab or a NUL when *-z* option is used.
12. path for "src"

13.a tab or a NUL when *-z* option is used; only exists for C or R.

14.path for "dst"; only exists for C or R.

15.an LF or a NUL when *-z* option is used, to terminate the record.

Possible status letters are:

- A: addition of a file
- C: copy of a file into a new one
- D: deletion of a file
- M: modification of the contents or mode of a file
- R: renaming of a file
- T: change in the type of the file (regular file, symbolic link or submodule)
- U: file is unmerged (you must complete the merge before it can be committed)
- X: "unknown" change type (most probably a bug, please report it)

Status letters C and R are always followed by a score (denoting the percentage of similarity between the source and target of the move or copy). Status letter M may be followed by a score (denoting the percentage of dissimilarity) for file rewrites.

The sha1 for "dst" is shown as all 0's if a file on the filesystem is out of sync with the index.

Example:

```
:100644 100644 5be4a4a 0000000 M file.c
```

Without the *-z* option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). Using *-z* the filename is output verbatim and the line is terminated by a NUL byte.

diff format for merges

"git-diff-tree", "git-diff-files" and "git-diff --raw" can take *-c* or *--cc* option to generate diff output also for merge commits. The output differs from the format described above in the following way:

1. there is a colon for each parent
2. there are more "src" modes and "src" sha1
3. status is concatenated status characters for each parent
4. no optional "score" number
5. tab-separated pathname(s) of the file

For *-c* and *--cc*, only the destination or final path is shown even if the file was renamed on any side of history. With *--combined-all-paths*, the name of the path in each parent is shown followed by the name of the path in the merge commit.

Examples for *-c* and *--cc* without *--combined-all-paths*:

```
::100644 100644 100644 fabadb8 cc95eb0 4866510 MM       desc.c
::100755 100755 100755 52b7a2d 6d1ac04 d2ac7d7 RM       bar.sh
::100644 100644 100644 e07d6c5 9042e82 ee91881 RR       phooey.c
```

Examples when *--combined-all-paths* added to either *-c* or *--cc*:

```
::100644 100644 100644 fabadb8 cc95eb0 4866510 MM       desc.c  desc.c
 desc.c
::100755 100755 100755 52b7a2d 6d1ac04 d2ac7d7 RM       foo.sh  bar.sh
 bar.sh
::100644 100644 100644 e07d6c5 9042e82 ee91881 RR       fooey.c fuey.c
 phooey.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patch text with -p

Running Section G.3.44, “git-diff(1)”, Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, Section G.3.42, “git-diff-index(1)”, Section G.3.43, “git-diff-tree(1)”, or Section G.3.41, “git-diff-files(1)” with the *-p* option produces patch text. You can customize the creation of patch text via the *GIT_EXTERNAL_DIFF* and the *GIT_DIFF_OPTS* environment variables (see Section G.3.1, “git(1)”), and the *diff* attribute (see Section G.4.2, “gitattributes(5)”).

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded by a "git diff" header that looks like this:

   ```
   diff --git a/file1 b/file2
   ```

   The *a/* and *b/* filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, */dev/null* is *not* used in place of the *a/* or *b/* filenames.

   When a rename/copy is involved, *file1* and *file2* show the name of the source file of the rename/copy and the name of the file that the rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

   ```
   old mode <mode>
   new mode <mode>
   deleted file mode <mode>
   new file mode <mode>
   copy from <path>
   copy to <path>
   rename from <path>
   rename to <path>
   similarity index <number>
   dissimilarity index <number>
   index <hash>..<hash> <mode>
   ```

   File modes are printed as 6-digit octal numbers including the file type and file permission bits.

   Path names in extended headers do not include the *a/* and *b/* prefixes.

   The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

   The index line includes the blob object names before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. Pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

4. All the *file1* files in the output refer to files before the commit, and all the *file2* files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

5. Hunk headers mention the name of the function to which the hunk applies. See "Defining a custom hunk-header" in Section G.4.2, “gitattributes(5)” for details of how to tailor this to specific languages.

Combined diff format

Any diff-generating command can take the *-c* or *--cc* option to produce a *combined diff* when showing a merge. This is the default format when showing merges with Section G.3.44, “git-diff(1)” or Section G.3.134, “git-show(1)”. Note also that you can give suitable *--diff-merges* option to any of these commands to force generation of diffs in a specific format.

A "combined diff" format looks like this:

```
diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
        return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +      unsigned char sha1[20];
 +      struct commit *cmit;
        struct commit_list *list;
        static int initialized = 0;
        struct commit_name *n;

 +      if (get_sha1(arg, sha1) < 0)
 +              usage(describe_usage);
 +      cmit = lookup_commit_reference(sha1);
 +      if (!cmit)
 +              usage(describe_usage);
 +
        if (!initialized) {
                initialized = 1;
                for_each_ref(get_name);
```

1. It is preceded by a "git diff" header, that looks like this (when the *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when the *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index <hash>,<hash>..<hash>
   mode <mode>,<mode>..<mode>
   ```

```
new file mode <mode>
deleted file mode <mode>,<mode>
```

The *mode <mode>,<mode>..<mode>* line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected content movement (renames and copying detection) are designed to work with the diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by a two-line from-file/to-file header:

```
--- a/file
+++ b/file
```

Similar to the two-line header for the traditional *unified* diff format, */dev/null* is used to signal created or deleted files.

However, if the --combined-all-paths option is provided, instead of a two-line from-file/to-file, you get an N +1 line from-file/to-file header, where N is the number of parents in the merge commit:

```
--- a/file
--- a/file
--- a/file
+++ b/file
```

This extended format can be useful if rename or copy detection is active, to allow you to see the original name of the file in different parents.

4. Chunk header format is modified to prevent people from accidentally feeding it to *patch -p1*. Combined diff format was created for review of merge commit changes, and was not meant to be applied. The change is similar to the change in the extended *index* header:

```
@@@ <from-file-range> <from-file-range> <to-file-range> @@@
```

There are (number of parents + 1) @ characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has - (minus -- appears in A but removed in B), + (plus -- missing in A but added to B), or " " (space -- unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X's line is different from it.

A - character in the column N means that the line appears in fileN but it does not appear in the result. A + character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two - removals from both file1 and file2, plus ++ to mean one line that was added does not appear in either file1 or file2). Also, eight other lines are the same from file1 but do not appear in file2 (hence prefixed with +).

When shown by *git diff-tree -c*, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by *git diff-files -c*, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

other diff formats

The *--summary* option describes newly added, deleted, renamed and copied files. The *--stat* option adds diffstat(1) graph to the output. These options can be combined with other options, such as *-p*, and are meant for human consumption.

When showing a change that involves a rename or a copy, *--stat* output formats the pathnames compactly by combining common prefix and suffix of the pathnames. For example, a change that moves *arch/i386/Makefile* to *arch/x86/Makefile* while modifying 4 lines will be shown like this:

```
arch/{i386 => x86}/Makefile    |   4 +--
```

The *--numstat* option gives the diffstat(1) information but is designed for easier machine consumption. An entry in *--numstat* output looks like this:

```
1       2       README
3       1       arch/{i386 => x86}/Makefile
```

That is, from left to right:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. pathname (possibly with rename/copy information);
6. a newline.

When *-z* output option is in effect, the output is formatted this way:

```
1       2       README NUL
3       1       NUL arch/i386/Makefile NUL arch/x86/Makefile NUL
```

That is:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. a NUL (only exists if renamed/copied);
6. pathname in preimage;
7. a NUL (only exists if renamed/copied);
8. pathname in postimage (only exists if renamed/copied);
9. a NUL.

The extra *NUL* before the preimage path in renamed case is to allow scripts that read the output to tell if the current record being read is a single-path record or a rename/copy record without reading ahead. After reading added and deleted lines, reading up to *NUL* would yield the pathname, but if that is *NUL*, the record will show two paths.

EXAMPLES

Various ways to check your working tree

```
$ git diff            ❶
$ git diff --cached   ❷
$ git diff HEAD       ❸
$ git diff AUTO_MERGE ❹
```

❶ Changes in the working tree not yet staged for the next commit.
❷ Changes between the index and your last commit; what you would be committing if you run *git commit* without *-a* option.

❸ Changes in the working tree since your last commit; what you would be committing if you run *git commit -a*
❹ Changes in the working tree you've made to resolve textual conflicts so far.

Comparing with arbitrary commits

```
$ git diff test            ❶
$ git diff HEAD -- ./test  ❷
$ git diff HEAD^ HEAD      ❸
```

❶ Instead of using the tip of the current branch, compare with the tip of "test" branch.
❷ Instead of comparing with the tip of "test" branch, compare with the tip of the current branch, but limit the comparison to the file "test".
❸ Compare the version before the last commit and the last commit.

Comparing branches

```
$ git diff topic master    ❶
$ git diff topic..master   ❷
$ git diff topic...master  ❸
```

❶ Changes between the tips of the topic and the master branches.
❷ Same as above.
❸ Changes that occurred on the master branch since when the topic branch was started off it.

Limiting the diff output

```
$ git diff --diff-filter=MRC            ❶
$ git diff --name-status                ❷
$ git diff arch/i386 include/asm-i386   ❸
```

❶ Show only modification, rename, and copy, but not addition or deletion.
❷ Show only names and the nature of change, but not actual diff output.
❸ Limit diff output to named subtrees.

Munging the diff output

```
$ git diff --find-copies-harder -B -C  ❶
$ git diff -R                          ❷
```

❶ Spend extra cycles to find renames, copies and complete rewrites (very expensive).
❷ Output diff in reverse.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

diff.autoRefreshIndex

When using *git diff* to compare with work tree files, do not consider stat-only changes as changed. Instead, silently run *git update-index --refresh* to update the cached stat information for paths whose contents in the work tree match the contents in the index. This option defaults to true. Note that this affects only *git diff* Porcelain, and not lower level *diff* commands such as *git diff-files*.

diff.dirstat

A comma separated list of *--dirstat* parameters specifying the default behavior of the *--dirstat* option to Section G.3.44, “git-diff(1)” and friends. The defaults can be overridden on the command line (using *--dirstat=<param1,param2,...>*). The fallback defaults (when not changed by *diff.dirstat*) are *changes,noncumulative,3*. The following parameters are available:

changes

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

lines

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive *--dirstat* behavior than the *changes* behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other *--*stat* options.

files

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest *--dirstat* behavior, since it does not have to look at the file contents at all.

cumulative

Count changes in a child directory for the parent directory as well. Note that when using *cumulative*, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the *noncumulative* parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: *files,10,cumulative*.

diff.statNameWidth

Limit the width of the filename part in --stat output. If set, applies to all commands generating --stat output except format-patch.

diff.statGraphWidth

Limit the width of the graph part in --stat output. If set, applies to all commands generating --stat output except format-patch.

diff.context

Generate diffs with <n> lines of context instead of the default of 3. This value is overridden by the -U option.

diff.interHunkContext

Show the context between diff hunks, up to the specified number of lines, thereby fusing the hunks that are close to each other. This value serves as the default for the *--inter-hunk-context* command line option.

diff.external

If this config variable is set, diff generation is not performed using the internal diff machinery, but using the given command. Can be overridden with the GIT_EXTERNAL_DIFF environment variable. The command is called with parameters as described under "git Diffs" in Section G.3.1, “git(1)”. Note: if you want to use an external diff program only on a subset of your files, you might want to use Section G.4.2, “gitattributes(5)” instead.

diff.ignoreSubmodules

Sets the default value of --ignore-submodules. Note that this affects only *git diff* Porcelain, and not lower level *diff* commands such as *git diff-files*. *git checkout* and *git switch* also honor this setting when reporting uncommitted changes. Setting it to *all* disables the submodule summary normally shown by *git commit* and *git status* when *status.submoduleSummary* is set unless it is overridden by using the --ignore-submodules command-line option. The *git submodule* commands are not affected by this setting. By default this is set to untracked so that any untracked submodules are ignored.

diff.mnemonicPrefix

If set, *git diff* uses a prefix pair that is different from the standard "a/" and "b/" depending on what is being compared. When this configuration is in effect, reverse diff output also swaps the order of the prefixes:

git diff

compares the (i)ndex and the (w)ork tree;

git diff HEAD

compares a (c)ommit and the (w)ork tree;

git diff --cached

compares a (c)ommit and the (i)ndex;

git diff HEAD:file1 file2

compares an (o)bject and a (w)ork tree entity;

git diff --no-index a b

compares two non-git things (1) and (2).

diff.noPrefix

If set, *git diff* does not show any source or destination prefix.

diff.srcPrefix

If set, *git diff* uses this source prefix. Defaults to "a/".

diff.dstPrefix

If set, *git diff* uses this destination prefix. Defaults to "b/".

diff.relative

If set to *true*, *git diff* does not show changes outside of the directory and show pathnames relative to the current directory.

diff.orderFile

File indicating how to order files within a diff. See the *-O* option to Section G.3.44, “git-diff(1)” for details. If *diff.orderFile* is a relative pathname, it is treated as relative to the top of the working tree.

diff.renameLimit

The number of files to consider in the exhaustive portion of copy/rename detection; equivalent to the *git diff* option *-l*. If not set, the default value is currently 1000. This setting has no effect if rename detection is turned off.

diff.renames

Whether and how Git detects renames. If set to "false", rename detection is disabled. If set to "true", basic rename detection is enabled. If set to "copies" or "copy", Git will detect copies, as well. Defaults to true. Note that this affects only *git diff* Porcelain like Section G.3.44, “git-diff(1)” and Section G.3.74, “git-log(1)”, and not lower level commands such as Section G.3.41, “git-diff-files(1)”.

diff.suppressBlankEmpty

A boolean to inhibit the standard behavior of printing a space before each empty output line. Defaults to false.

diff.submodule

Specify the format in which differences in submodules are shown. The "short" format just shows the names of the commits at the beginning and end of the range. The "log" format lists the commits in the range like Section G.3.141, “git-submodule(1)” *summary* does. The "diff" format shows an inline diff of the changed contents of the submodule. Defaults to "short".

diff.wordRegex

A POSIX Extended Regular Expression used to determine what is a "word" when performing word-by-word difference calculations. Character sequences that match the regular expression are "words", all other characters are **ignorable** whitespace.

diff.<driver>.command

The custom diff driver command. See Section G.4.2, “gitattributes(5)” for details.

diff.<driver>.xfuncname

The regular expression that the diff driver should use to recognize the hunk header. A built-in pattern may also be used. See Section G.4.2, “gitattributes(5)” for details.

diff.<driver>.binary

Set this option to true to make the diff driver treat files as binary. See Section G.4.2, “gitattributes(5)” for details.

diff.<driver>.textconv

The command that the diff driver should call to generate the text-converted version of a file. The result of the conversion is used to generate a human-readable diff. See Section G.4.2, “gitattributes(5)” for details.

diff.<driver>.wordRegex

The regular expression that the diff driver should use to split words in a line. See Section G.4.2, “gitattributes(5)” for details.

diff.<driver>.cachetextconv

Set this option to true to make the diff driver cache the text conversion outputs. See Section G.4.2, “gitattributes(5)” for details.

araxis

Use Araxis Merge (requires a graphical session)

bc

Use Beyond Compare (requires a graphical session)

bc3

Use Beyond Compare (requires a graphical session)

bc4

Use Beyond Compare (requires a graphical session)

codecompare

Use Code Compare (requires a graphical session)

deltawalker

Use DeltaWalker (requires a graphical session)

diffmerge

Use DiffMerge (requires a graphical session)

diffuse

Use Diffuse (requires a graphical session)

ecmerge

Use ECMerge (requires a graphical session)

emerge

Use Emacs' Emerge

examdiff

Use ExamDiff Pro (requires a graphical session)

guiffy

Use Guiffy's Diff Tool (requires a graphical session)

gvimdiff

Use gVim (requires a graphical session)

kdiff3

Use KDiff3 (requires a graphical session)

kompare

Use Kompare (requires a graphical session)

meld

Use Meld (requires a graphical session)

nvimdiff

Use Neovim

opendiff

Use FileMerge (requires a graphical session)

p4merge

Use HelixCore P4Merge (requires a graphical session)

smerge

Use Sublime Merge (requires a graphical session)

tkdiff

Use TkDiff (requires a graphical session)

vimdiff

Use Vim

winmerge

Use WinMerge (requires a graphical session)

xxdiff

Use xxdiff (requires a graphical session)

diff.indentHeuristic

Set this option to *false* to disable the default heuristics that shift diff hunk boundaries to make patches easier to read.

diff.algorithm

Choose a diff algorithm. The variants are as follows:

default, *myers*

The basic greedy diff algorithm. Currently, this is the default.

minimal

Spend extra time to make sure the smallest possible diff is produced.

patience

Use "patience diff" algorithm when generating patches.

histogram

This algorithm extends the patience algorithm to "support low-occurrence common elements".

diff.wsErrorHighlight

Highlight whitespace errors in the *context*, *old* or *new* lines of the diff. Multiple values are separated by comma, *none* resets previous values, *default* reset the list to *new* and *all* is a shorthand for *old,new,context*. The white-space errors are colored with *color.diff.whitespace*. The command line option *--ws-error-highlight=<kind>* overrides this setting.

diff.colorMoved

If set to either a valid *<mode>* or a true value, moved lines in a diff are colored differently, for details of valid modes see *--color-moved* in Section G.3.44, “git-diff(1)”. If simply set to true the default color mode will be used. When set to false, moved lines are not colored.

diff.colorMovedWS

When moved lines are colored using e.g. the *diff.colorMoved* setting, this option controls the *<mode>* how spaces are treated. For details of valid modes see *--color-moved-ws* in Section G.3.44, “git-diff(1)”.

SEE ALSO

diff(1), Section G.3.45, “git-difftool(1)”, Section G.3.74, “git-log(1)”, Section G.4.4, “gitdiffcore(7)”, Section G.3.54, “git-format-patch(1)”, Section G.3.5, “git-apply(1)”, Section G.3.134, “git-show(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.45. git-difftool(1)

2024-07-02

Revision History

2024-07-02

NAME

git-difftool - Show changes using common diff tools

SYNOPSIS

git difftool [<options>] [<commit> [<commit>]] [--] [<path>…]

DESCRIPTION

git difftool is a Git command that allows you to compare and edit files between revisions using common diff tools. *git difftool* is a frontend to *git diff* and accepts the same options and arguments. See Section G.3.44, “git-diff(1)”.

OPTIONS

-d , --dir-diff

Copy the modified files to a temporary location and perform a directory diff on them. This mode never prompts before launching the diff tool.

-y , --no-prompt

Do not prompt before launching a diff tool.

--prompt

Prompt before each invocation of the diff tool. This is the default behaviour; the option is provided to override any configuration settings.

--rotate-to=<file>

Start showing the diff for the given path, the paths before it will move to the end and output.

--skip-to=<file>

Start showing the diff for the given path, skipping all the paths before it.

-t <tool> , --tool=<tool>

Use the diff tool specified by <tool>. Valid values include emerge, kompare, meld, and vimdiff. Run *git difftool --tool-help* for the list of valid <tool> settings.

If a diff tool is not specified, *git difftool* will use the configuration variable *diff.tool*. If the configuration variable *diff.tool* is not set, *git difftool* will pick a suitable default.

You can explicitly provide a full path to the tool by setting the configuration variable *difftool.<tool>.path*. For example, you can configure the absolute path to kdiff3 by setting *difftool.kdiff3.path*. Otherwise, *git difftool* assumes the tool is available in PATH.

Instead of running one of the known diff tools, *git difftool* can be customized to run an alternative program by specifying the command line to invoke in a configuration variable *difftool.<tool>.cmd*.

When *git difftool* is invoked with this tool (either through the *-t* or *--tool* option or the *diff.tool* configuration variable) the configured command line will be invoked with the following variables available: *$LOCAL* is set to the name of the temporary file containing the contents of the diff pre-image and *$REMOTE* is set to the name of the temporary file containing the contents of the diff post-image. *$MERGED* is the name of the file which is being compared. *$BASE* is provided for compatibility with custom merge tool commands and has the same value as *$MERGED*.

--tool-help

Print a list of diff tools that may be used with *--tool*.

--[no-]symlinks

git difftool's default behavior is to create symlinks to the working tree when run in *--dir-diff* mode and the right-hand side of the comparison yields the same content as the file in the working tree.

Specifying *--no-symlinks* instructs *git difftool* to create copies instead. *--no-symlinks* is the default on Windows.

-x <command> , --extcmd=<command>

Specify a custom command for viewing diffs. *git-difftool* ignores the configured defaults and runs *<command> $LOCAL $REMOTE* when this option is specified. Additionally, *$BASE* is set in the environment.

-g , --[no-]gui

When *git-difftool* is invoked with the *-g* or *--gui* option the default diff tool will be read from the configured *diff.guitool* variable instead of *diff.tool*. This may be selected automatically using the configuration variable *difftool.guiDefault*. The *--no-gui* option can be used to override these settings. If *diff.guitool* is not set, we will fallback in the order of *merge.guitool*, *diff.tool*, *merge.tool* until a tool is found.

--[no-]trust-exit-code

Errors reported by the diff tool are ignored by default. Use *--trust-exit-code* to make *git-difftool* exit when an invoked diff tool returns a non-zero exit code.

git-difftool will forward the exit code of the invoked tool when *--trust-exit-code* is used.

See Section G.3.44, “git-diff(1)” for the full list of supported options.

CONFIGURATION

git difftool falls back to *git mergetool* config variables when the difftool equivalents have not been defined.

Everything above this line in this section isn't included from the Section G.3.29, “git-config(1)” documentation. The content that follows is the same as what's found there:

diff.tool

Controls which diff tool is used by Section G.3.45, “git-difftool(1)”. This variable overrides the value configured in *merge.tool*. The list below shows the valid built-in values. Any other value is treated as a custom diff tool and requires that a corresponding difftool.<tool>.cmd variable is defined.

diff.guitool

Controls which diff tool is used by Section G.3.45, “git-difftool(1)” when the -g/--gui flag is specified. This variable overrides the value configured in *merge.guitool*. The list below shows the valid built-in values. Any other value is treated as a custom diff tool and requires that a corresponding difftool.<guitool>.cmd variable is defined.

difftool.<tool>.cmd

Specify the command to invoke the specified diff tool. The specified command is evaluated in shell with the following variables available: *LOCAL* is set to the name of the temporary file containing the contents of the diff pre-image and *REMOTE* is set to the name of the temporary file containing the contents of the diff post-image.

See the *--tool=<tool>* option in Section G.3.45, "git-difftool(1)" for more details.

difftool.<tool>.path

Override the path for the given tool. This is useful in case your tool is not in the PATH.

difftool.trustExitCode

Exit difftool if the invoked diff tool returns a non-zero exit status.

See the *--trust-exit-code* option in Section G.3.45, "git-difftool(1)" for more details.

difftool.prompt

Prompt before each invocation of the diff tool.

difftool.guiDefault

Set *true* to use the *diff.guitool* by default (equivalent to specifying the *--gui* argument), or *auto* to select *diff.guitool* or *diff.tool* depending on the presence of a *DISPLAY* environment variable value. The default is *false*, where the *--gui* argument must be provided explicitly for the *diff.guitool* to be used.

SEE ALSO

Section G.3.44, "git-diff(1)"

Show changes between commits, commit and working tree, etc

Section G.3.88, "git-mergetool(1)"

Run merge conflict resolution tools to resolve merge conflicts

Section G.3.29, "git-config(1)"

Get and set repository or global options

GIT

Part of the Section G.3.1, "git(1)" suite

G.3.46. git-fast-export(1)

2024-07-02

Revision History

2024-07-02

NAME

git-fast-export - Git data exporter

SYNOPSIS

git fast-export [<options>] | *git fast-import*

DESCRIPTION

This program dumps the given revisions in a form suitable to be piped into *git fast-import*.

You can use it as a human-readable bundle replacement (see Section G.3.12, “git-bundle(1)”), or as a format that can be edited before being fed to *git fast-import* in order to do history rewrites (an ability relied on by tools like *git filter-repo*).

OPTIONS

--progress=<n>

Insert *progress* statements every <n> objects, to be shown by *git fast-import* during import.

--signed-tags=(verbatim|warn|warn-strip|strip|abort)

Specify how to handle signed tags. Since any transformation after the export can change the tag names (which can also happen when excluding revisions) the signatures will not match.

When asking to *abort* (which is the default), this program will die when encountering a signed tag. With *strip*, the tags will silently be made unsigned, with *warn-strip* they will be made unsigned but a warning will be displayed, with *verbatim*, they will be silently exported and with *warn*, they will be exported, but you will see a warning.

--tag-of-filtered-object=(abort|drop|rewrite)

Specify how to handle tags whose tagged object is filtered out. Since revisions and files to export can be limited by path, tagged objects may be filtered completely.

When asking to *abort* (which is the default), this program will die when encountering such a tag. With *drop* it will omit such tags from the output. With *rewrite*, if the tagged object is a commit, it will rewrite the tag to tag an ancestor commit (via parent rewriting; see Section G.3.120, “git-rev-list(1)”).

-M , -C

Perform move and/or copy detection, as described in the Section G.3.44, “git-diff(1)” manual page, and use it to generate rename and copy commands in the output dump.

Note that earlier versions of this command did not complain and produced incorrect results if you gave these options.

--export-marks=<file>

Dumps the internal marks table to <file> when complete. Marks are written one per line as *:markid SHA-1*. Only marks for revisions are dumped; marks for blobs are ignored. Backends can use this file to validate imports after they have been completed, or to save the marks table across incremental runs. As <file> is only opened and truncated at completion, the same path can also be safely given to --import-marks. The file will not be written if no new object has been marked/exported.

--import-marks=<file>

Before processing any input, load the marks specified in <file>. The input file must exist, must be readable, and must use the same format as produced by --export-marks.

--mark-tags

In addition to labelling blobs and commits with mark ids, also label tags. This is useful in conjunction with *--export-marks* and *--import-marks*, and is also useful (and necessary) for exporting of nested tags. It does not hurt other cases and would be the default, but many fast-import frontends are not prepared to accept tags with mark identifiers.

Any commits (or tags) that have already been marked will not be exported again. If the backend uses a similar --import-marks file, this allows for incremental bidirectional exporting of the repository by keeping the marks the same across runs.

--fake-missing-tagger

Some old repositories have tags without a tagger. The fast-import protocol was pretty strict about that, and did not allow that. So fake a tagger to be able to fast-import the output.

--use-done-feature

Start the stream with a *feature done* stanza, and terminate it with a *done* command.

--no-data

Skip output of blob objects and instead refer to blobs via their original SHA-1 hash. This is useful when rewriting the directory structure or history of a repository without touching the contents of individual files. Note that the resulting stream can only be used by a repository which already contains the necessary objects.

--full-tree

This option will cause fast-export to issue a "deleteall" directive for each commit followed by a full list of all files in the commit (as opposed to just listing the files which are different from the commit's first parent).

--anonymize

Anonymize the contents of the repository while still retaining the shape of the history and stored tree. See the section on *ANONYMIZING* below.

--anonymize-map=<from>[:<to>]

Convert token *<from>* to *<to>* in the anonymized output. If *<to>* is omitted, map *<from>* to itself (i.e., do not anonymize it). See the section on *ANONYMIZING* below.

--reference-excluded-parents

By default, running a command such as *git fast-export master~5..master* will not include the commit master~5 and will make master~4 no longer have master~5 as a parent (though both the old master~4 and new master~4 will have all the same files). Use --reference-excluded-parents to instead have the stream refer to commits in the excluded range of history by their sha1sum. Note that the resulting stream can only be used by a repository which already contains the necessary parent commits.

--show-original-ids

Add an extra directive to the output for commits and blobs, *original-oid <SHA1SUM>*. While such directives will likely be ignored by importers such as git-fast-import, it may be useful for intermediary filters (e.g. for rewriting commit messages which refer to older commits, or for stripping blobs by id).

--reencode=(yes|no|abort)

Specify how to handle *encoding* header in commit objects. When asking to *abort* (which is the default), this program will die when encountering such a commit object. With *yes*, the commit message will be re-encoded into UTF-8. With *no*, the original encoding will be preserved.

--refspec

Apply the specified refspec to each ref exported. Multiple of them can be specified.

[<git-rev-list-args>…]

A list of arguments, acceptable to *git rev-parse* and *git rev-list*, that specifies the specific objects and references to export. For example, *master~10..master* causes the current master reference to be exported along with all objects added since its 10th ancestor commit and (unless the --reference-excluded-parents option is specified) all files common to master~9 and master~10.

EXAMPLES

```
$ git fast-export --all | (cd /empty/repository && git fast-import)
```

This will export the whole repository and import it into the existing empty repository. Except for reencoding commits that are not in UTF-8, it would be a one-to-one mirror.

```
$ git fast-export master~5..master |
        sed "s|refs/heads/master|refs/heads/other|" |
        git fast-import
```

This makes a new branch called *other* from *master~5..master* (i.e. if *master* has linear history, it will take the last 5 commits).

Note that this assumes that none of the blobs and commit messages referenced by that revision range contains the string *refs/heads/master*.

ANONYMIZING

If the *--anonymize* option is given, git will attempt to remove all identifying information from the repository while still retaining enough of the original tree and history patterns to reproduce some bugs. The goal is that a git bug which is found on a private repository will persist in the anonymized repository, and the latter can be shared with git developers to help solve the bug.

With this option, git will replace all refnames, paths, blob contents, commit and tag messages, names, and email addresses in the output with anonymized data. Two instances of the same string will be replaced equivalently (e.g., two commits with the same author will have the same anonymized author in the output, but bear no resemblance to the original author string). The relationship between commits, branches, and tags is retained, as well as the commit timestamps (but the commit messages and refnames bear no resemblance to the originals). The relative makeup of the tree is retained (e.g., if you have a root tree with 10 files and 3 trees, so will the output), but their names and the contents of the files will be replaced.

If you think you have found a git bug, you can start by exporting an anonymized stream of the whole repository:

```
$ git fast-export --anonymize --all >anon-stream
```

Then confirm that the bug persists in a repository created from that stream (many bugs will not, as they really do depend on the exact repository contents):

```
$ git init anon-repo
$ cd anon-repo
$ git fast-import <../anon-stream
$ ... test your bug ...
```

If the anonymized repository shows the bug, it may be worth sharing *anon-stream* along with a regular bug report. Note that the anonymized stream compresses very well, so gzipping it is encouraged. If you want to examine the stream to see that it does not contain any private data, you can peruse it directly before sending. You may also want to try:

```
$ perl -pe 's/\d+/X/g' <anon-stream | sort -u | less
```

which shows all of the unique lines (with numbers converted to "X", to collapse "User 0", "User 1", etc into "User X"). This produces a much smaller output, and it is usually easy to quickly confirm that there is no private data in the stream.

Reproducing some bugs may require referencing particular commits or paths, which becomes challenging after refnames and paths have been anonymized. You can ask for a particular token to be left as-is or mapped to a new value. For example, if you have a bug which reproduces with *git rev-list sensitive -- secret.c*, you can run:

```
$ git fast-export --anonymize --all \
      --anonymize-map=sensitive:foo \
      --anonymize-map=secret.c:bar.c \
```

```
>stream
```

After importing the stream, you can then run *git rev-list foo -- bar.c* in the anonymized repository.

Note that paths and refnames are split into tokens at slash boundaries. The command above would anonymize *subdir/secret.c* as something like *path123/bar.c*; you could then search for *bar.c* in the anonymized repository to determine the final pathname.

To make referencing the final pathname simpler, you can map each path component; so if you also anonymize *subdir* to *publicdir*, then the final pathname would be *publicdir/bar.c*.

LIMITATIONS

Since *git fast-import* cannot tag trees, you will not be able to export the linux.git repository completely, as it contains a tag referencing a tree instead of a commit.

SEE ALSO

Section G.3.47, “git-fast-import(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.47. git-fast-import(1)

2024-07-02
Revision History
2024-07-02

NAME

git-fast-import - Backend for fast Git data importers

SYNOPSIS

frontend | *git fast-import* [<options>]

DESCRIPTION

This program is usually not what the end user wants to run directly. Most end users want to use one of the existing frontend programs, which parses a specific type of foreign source and feeds the contents stored there to *git fast-import*.

fast-import reads a mixed command/data stream from standard input and writes one or more packfiles directly into the current repository. When EOF is received on standard input, fast import writes out updated branch and tag refs, fully updating the current repository with the newly imported data.

The fast-import backend itself can import into an empty repository (one that has already been initialized by *git init*) or incrementally update an existing populated repository. Whether or not incremental imports are supported from a particular foreign source depends on the frontend program in use.

OPTIONS

--force

Force updating modified existing branches, even if doing so would cause commits to be lost (as the new commit does not contain the old commit).

--quiet

Disable the output shown by --stats, making fast-import usually be silent when it is successful. However, if the import stream has directives intended to show user output (e.g. *progress* directives), the corresponding messages will still be shown.

--stats

Display some basic statistics about the objects fast-import has created, the packfiles they were stored into, and the memory used by fast-import during this run. Showing this output is currently the default, but can be disabled with --quiet.

--allow-unsafe-features

Many command-line options can be provided as part of the fast-import stream itself by using the *feature* or *option* commands. However, some of these options are unsafe (e.g., allowing fast-import to access the filesystem outside of the repository). These options are disabled by default, but can be allowed by providing this option on the command line. This currently impacts only the *export-marks*, *import-marks*, and *import-marks-if-exists* feature commands.

```
Only enable this option if you trust the program generating the
fast-import stream! This option is enabled automatically for
remote-helpers that use the `import` capability, as they are
already trusted to run their own code.
```

1. Options for Frontends

--cat-blob-fd=<fd>

Write responses to *get-mark*, *cat-blob*, and *ls* queries to the file descriptor <fd> instead of *stdout*. Allows *progress* output intended for the end-user to be separated from other output.

--date-format=<fmt>

Specify the type of dates the frontend will supply to fast-import within *author*, *committer* and *tagger* commands. See Date Formats below for details about which formats are supported, and their syntax.

--done

Terminate with error if there is no *done* command at the end of the stream. This option might be useful for detecting errors that cause the frontend to terminate before it has started to write a stream.

2. Locations of Marks Files

--export-marks=<file>

Dumps the internal marks table to <file> when complete. Marks are written one per line as *:markid SHA-1*. Frontends can use this file to validate imports after they have been completed, or to save the marks table across incremental runs. As <file> is only opened and truncated at checkpoint (or completion) the same path can also be safely given to --import-marks.

--import-marks=<file>

Before processing any input, load the marks specified in <file>. The input file must exist, must be readable, and must use the same format as produced by --export-marks. Multiple options may be supplied to import more than one set of marks. If a mark is defined to different values, the last file wins.

--import-marks-if-exists=<file>

Like --import-marks but instead of erroring out, silently skips the file if it does not exist.

--[no-]relative-marks

After specifying --relative-marks the paths specified with --import-marks= and --export-marks= are relative to an internal directory in the current repository. In git-fast-import this means that the paths are relative to the .git/info/fast-import directory. However, other importers may use a different location.

Relative and non-relative marks may be combined by interweaving --(no-)-relative-marks with the --(import|export)-marks= options.

3. Submodule Rewriting

--rewrite-submodules-from=<name>:<file> , --rewrite-submodules-to=<name>:<file>

Rewrite the object IDs for the submodule specified by <name> from the values used in the from <file> to those used in the to <file>. The from marks should have been created by *git fast-export*, and the to marks should have been created by *git fast-import* when importing that same submodule.

<name> may be any arbitrary string not containing a colon character, but the same value must be used with both options when specifying corresponding marks. Multiple submodules may be specified with different values for <name>. It is an error not to use these options in corresponding pairs.

These options are primarily useful when converting a repository from one hash algorithm to another; without them, fast-import will fail if it encounters a submodule because it has no way of writing the object ID into the new hash algorithm.

4. Performance and Compression Tuning

--active-branches=<n>

Maximum number of branches to maintain active at once. See Memory Utilization below for details. Default is 5.

--big-file-threshold=<n>

Maximum size of a blob that fast-import will attempt to create a delta for, expressed in bytes. The default is 512m (512 MiB). Some importers may wish to lower this on systems with constrained memory.

--depth=<n>

Maximum delta depth, for blob and tree deltification. Default is 50.

--export-pack-edges=<file>

After creating a packfile, print a line of data to <file> listing the filename of the packfile and the last commit on each branch that was written to that packfile. This information may be useful after importing projects whose total object set exceeds the 4 GiB packfile limit, as these commits can be used as edge points during calls to *git pack-objects*.

--max-pack-size=<n>

Maximum size of each output packfile. The default is unlimited.

fastimport.unpackLimit

See Section G.3.29, “git-config(1)”

PERFORMANCE

The design of fast-import allows it to import large projects in a minimum amount of memory usage and processing time. Assuming the frontend is able to keep up with fast-import and feed it a constant stream of data, import times for projects holding 10+ years of history and containing 100,000+ individual commits are generally completed in just 1-2 hours on quite modest (~$2,000 USD) hardware.

Most bottlenecks appear to be in foreign source data access (the source just cannot extract revisions fast enough) or disk IO (fast-import writes as fast as the disk will take the data). Imports will run faster if the source data is stored on a different drive than the destination Git repository (due to less IO contention).

DEVELOPMENT COST

A typical frontend for fast-import tends to weigh in at approximately 200 lines of Perl/Python/Ruby code. Most developers have been able to create working importers in just a couple of hours, even though it is their first exposure to fast-import, and sometimes even to Git. This is an ideal situation, given that most conversion tools are throw-away (use once, and never look back).

PARALLEL OPERATION

Like *git push* or *git fetch*, imports handled by fast-import are safe to run alongside parallel *git repack -a -d* or *git gc* invocations, or any other Git operation (including *git prune*, as loose objects are never used by fast-import).

fast-import does not lock the branch or tag refs it is actively importing. After the import, during its ref update phase, fast-import tests each existing branch ref to verify the update will be a fast-forward update (the commit stored in the ref is contained in the new history of the commit to be written). If the update is not a fast-forward update, fast-import will skip updating that ref and instead prints a warning message. fast-import will always attempt to update all branch refs, and does not stop on the first failure.

Branch updates can be forced with --force, but it's recommended that this only be used on an otherwise quiet repository. Using --force is not necessary for an initial import into an empty repository.

TECHNICAL DISCUSSION

fast-import tracks a set of branches in memory. Any branch can be created or modified at any point during the import process by sending a *commit* command on the input stream. This design allows a frontend program to process an unlimited number of branches simultaneously, generating commits in the order they are available from the source data. It also simplifies the frontend programs considerably.

fast-import does not use or alter the current working directory, or any file within it. (It does however update the current Git repository, as referenced by *GIT_DIR*.) Therefore an import frontend may use the working directory for its own purposes, such as extracting file revisions from the foreign source. This ignorance of the working directory also allows fast-import to run very quickly, as it does not need to perform any costly file update operations when switching between branches.

INPUT FORMAT

With the exception of raw file data (which Git does not interpret) the fast-import input format is text (ASCII) based. This text based format simplifies development and debugging of frontend programs, especially when a higher level language such as Perl, Python or Ruby is being used.

fast-import is very strict about its input. Where we say SP below we mean **exactly** one space. Likewise LF means one (and only one) linefeed and HT one (and only one) horizontal tab. Supplying additional whitespace characters will cause unexpected results, such as branch names or file names with leading or trailing spaces in their name, or early termination of fast-import when it encounters unexpected input.

1. Stream Comments

To aid in debugging frontends fast-import ignores any line that begins with # (ASCII pound/hash) up to and including the line ending *LF*. A comment line may contain any sequence of bytes that does not contain an LF and therefore may be used to include any detailed debugging information that might be specific to the frontend and useful when inspecting a fast-import data stream.

2. Date Formats

The following date formats are supported. A frontend should select the format it will use for this import by passing the format name in the --date-format=<fmt> command-line option.

raw

This is the Git native format and is *<time> SP <offutc>*. It is also fast-import's default format, if --date-format was not specified.

The time of the event is specified by <*time*> as the number of seconds since the UNIX epoch (midnight, Jan 1, 1970, UTC) and is written as an ASCII decimal integer.

The local offset is specified by <*offutc*> as a positive or negative offset from UTC. For example EST (which is 5 hours behind UTC) would be expressed in <*tz*> by -0500 while UTC is +0000. The local offset does not affect <*time*>; it is used only as an advisement to help formatting routines display the timestamp.

If the local offset is not available in the source material, use +0000, or the most common local offset. For example many organizations have a CVS repository which has only ever been accessed by users who are located in the same location and time zone. In this case a reasonable offset from UTC could be assumed.

Unlike the *rfc2822* format, this format is very strict. Any variation in formatting will cause fast-import to reject the value, and some sanity checks on the numeric values may also be performed.

raw-permissive

This is the same as *raw* except that no sanity checks on the numeric epoch and local offset are performed. This can be useful when trying to filter or import an existing history with e.g. bogus timezone values.

rfc2822

This is the standard date format as described by RFC 2822.

An example value is Tue Feb 6 11:22:18 2007 -0500. The Git parser is accurate, but a little on the lenient side. It is the same parser used by *git am* when applying patches received from email.

Some malformed strings may be accepted as valid dates. In some of these cases Git will still be able to obtain the correct date from the malformed string. There are also some types of malformed strings which Git will parse wrong, and yet consider valid. Seriously malformed strings will be rejected.

Unlike the *raw* format above, the time zone/UTC offset information contained in an RFC 2822 date string is used to adjust the date value to UTC prior to storage. Therefore it is important that this information be as accurate as possible.

If the source material uses RFC 2822 style dates, the frontend should let fast-import handle the parsing and conversion (rather than attempting to do it itself) as the Git parser has been well tested in the wild.

Frontends should prefer the *raw* format if the source material already uses UNIX-epoch format, can be coaxed to give dates in that format, or its format is easily convertible to it, as there is no ambiguity in parsing.

now

Always use the current time and time zone. The literal *now* must always be supplied for <*when*>.

This is a toy format. The current time and time zone of this system is always copied into the identity string at the time it is being created by fast-import. There is no way to specify a different time or time zone.

This particular format is supplied as it's short to implement and may be useful to a process that wants to create a new commit right now, without needing to use a working directory or *git update-index*.

If separate *author* and *committer* commands are used in a *commit* the timestamps may not match, as the system clock will be polled twice (once for each command). The only way to ensure that both author and committer identity information has the same timestamp is to omit *author* (thus copying from *committer*) or to use a date format other than *now*.

3. Commands

fast-import accepts several commands to update the current repository and control the current import process. More detailed discussion (with examples) of each command follows later.

commit

Creates a new branch or updates an existing branch by creating a new commit and updating the branch to point at the newly created commit.

tag

Creates an annotated tag object from an existing commit or branch. Lightweight tags are not supported by this command, as they are not recommended for recording meaningful points in time.

reset

Reset an existing branch (or a new branch) to a specific revision. This command must be used to change a branch to a specific revision without making a commit on it.

blob

Convert raw file data into a blob, for future use in a *commit* command. This command is optional and is not needed to perform an import.

alias

Record that a mark refers to a given object without first creating any new object. Using --import-marks and referring to missing marks will cause fast-import to fail, so aliases can provide a way to set otherwise pruned commits to a valid value (e.g. the nearest non-pruned ancestor).

checkpoint

Forces fast-import to close the current packfile, generate its unique SHA-1 checksum and index, and start a new packfile. This command is optional and is not needed to perform an import.

progress

Causes fast-import to echo the entire line to its own standard output. This command is optional and is not needed to perform an import.

done

Marks the end of the stream. This command is optional unless the *done* feature was requested using the *--done* command-line option or *feature done* command.

get-mark

Causes fast-import to print the SHA-1 corresponding to a mark to the file descriptor set with *--cat-blob-fd*, or *stdout* if unspecified.

cat-blob

Causes fast-import to print a blob in *cat-file --batch* format to the file descriptor set with *--cat-blob-fd* or *stdout* if unspecified.

ls

Causes fast-import to print a line describing a directory entry in *ls-tree* format to the file descriptor set with *--cat-blob-fd* or *stdout* if unspecified.

feature

Enable the specified feature. This requires that fast-import supports the specified feature, and aborts if it does not.

option

Specify any of the options listed under OPTIONS that do not change stream semantic to suit the frontend's needs. This command is optional and is not needed to perform an import.

4. *commit*

Create or update a branch with a new commit, recording one logical change to the project.

```
        'commit' SP <ref> LF
        mark?
        original-oid?
        ('author' (SP <name>)? SP LT <email> GT SP <when> LF)?
        'committer' (SP <name>)? SP LT <email> GT SP <when> LF
        ('encoding' SP <encoding>)?
        data
        ('from' SP <commit-ish> LF)?
        ('merge' SP <commit-ish> LF)*
        (filemodify | filedelete | filecopy | filerename | filedeleteall |
 notemodify)*
        LF?
```

where <*ref*> is the name of the branch to make the commit on. Typically branch names are prefixed with *refs/heads/* in Git, so importing the CVS branch symbol *RELENG-1_0* would use *refs/heads/RELENG-1_0* for the value of <*ref*>. The value of <*ref*> must be a valid refname in Git. As *LF* is not valid in a Git refname, no quoting or escaping syntax is supported here.

A *mark* command may optionally appear, requesting fast-import to save a reference to the newly created commit for future use by the frontend (see below for format). It is very common for frontends to mark every commit they create, thereby allowing future branch creation from any imported commit.

The *data* command following *committer* must supply the commit message (see below for *data* command syntax). To import an empty commit message use a 0 length data. Commit messages are free-form and are not interpreted by Git. Currently they must be encoded in UTF-8, as fast-import does not permit other encodings to be specified.

Zero or more *filemodify*, *filedelete*, *filecopy*, *filerename*, *filedeleteall* and *notemodify* commands may be included to update the contents of the branch prior to creating the commit. These commands may be supplied in any order. However it is recommended that a *filedeleteall* command precede all *filemodify*, *filecopy*, *filerename* and *notemodify* commands in the same commit, as *filedeleteall* wipes the branch clean (see below).

The *LF* after the command is optional (it used to be required). Note that for reasons of backward compatibility, if the commit ends with a *data* command (i.e. it has no *from*, *merge*, *filemodify*, *filedelete*, *filecopy*, *filerename*, *filedeleteall* or *notemodify* commands) then two *LF* commands may appear at the end of the command instead of just one.

4.1. *author*

An *author* command may optionally appear, if the author information might differ from the committer information. If *author* is omitted then fast-import will automatically use the committer's information for the author portion of the commit. See below for a description of the fields in *author*, as they are identical to *committer*.

4.2. *committer*

The *committer* command indicates who made this commit, and when they made it.

Here <*name*> is the person's display name (for example Com M Itter) and <*email*> is the person's email address (cm@example.com). *LT* and *GT* are the literal less-than (\x3c) and greater-than (\x3e) symbols. These are required to delimit the email address from the other fields in the line. Note that <*name*> and <*email*> are free-form and may contain any sequence of bytes, except *LT*, *GT* and *LF*. <*name*> is typically UTF-8 encoded.

The time of the change is specified by <*when*> using the date format that was selected by the --date-format=<fmt> command-line option. See Date Formats above for the set of supported formats, and their syntax.

4.3. *encoding*

The optional *encoding* command indicates the encoding of the commit message. Most commits are UTF-8 and the encoding is omitted, but this allows importing commit messages into git without first reencoding them.

4.4. *from*

The *from* command is used to specify the commit to initialize this branch from. This revision will be the first ancestor of the new commit. The state of the tree built at this commit will begin with the state at the *from* commit, and be altered by the content modifications in this commit.

Omitting the *from* command in the first commit of a new branch will cause fast-import to create that commit with no ancestor. This tends to be desired only for the initial commit of a project. If the frontend creates all files from scratch when making a new branch, a *merge* command may be used instead of *from* to start the commit with an empty tree. Omitting the *from* command on existing branches is usually desired, as the current commit on that branch is automatically assumed to be the first ancestor of the new commit.

As *LF* is not valid in a Git refname or SHA-1 expression, no quoting or escaping syntax is supported within <*commit-ish*>.

Here <*commit-ish*> is any of the following:

- The name of an existing branch already in fast-import's internal branch table. If fast-import doesn't know the name, it's treated as a SHA-1 expression.
- A mark reference, *:*<*idnum*>, where <*idnum*> is the mark number.

  The reason fast-import uses *:* to denote a mark reference is this character is not legal in a Git branch name. The leading *:* makes it easy to distinguish between the mark 42 (*:42*) and the branch 42 (*42* or *refs/heads/42*), or an abbreviated SHA-1 which happened to consist only of base-10 digits.

  Marks must be declared (via *mark*) before they can be used.
- A complete 40 byte or abbreviated commit SHA-1 in hex.
- Any valid Git SHA-1 expression that resolves to a commit. See SPECIFYING REVISIONS in Section G.4.14, “gitrevisions(7)” for details.
- The special null SHA-1 (40 zeros) specifies that the branch is to be removed.

The special case of restarting an incremental import from the current branch value should be written as:

```
from refs/heads/branch^0
```

The *^0* suffix is necessary as fast-import does not permit a branch to start from itself, and the branch is created in memory before the *from* command is even read from the input. Adding *^0* will force fast-import to resolve the commit through Git's revision parsing library, rather than its internal branch table, thereby loading in the existing value of the branch.

4.5. *merge*

Includes one additional ancestor commit. The additional ancestry link does not change the way the tree state is built at this commit. If the *from* command is omitted when creating a new branch, the first *merge* commit will be the first ancestor of the current commit, and the branch will start out with no files. An unlimited number of *merge* commands per commit are permitted by fast-import, thereby establishing an n-way merge.

Here <*commit-ish*> is any of the commit specification expressions also accepted by *from* (see above).

4.6. *filemodify*

Included in a *commit* command to add a new file or change the content of an existing file. This command has two different means of specifying the content of the file.

External data format

The data content for the file was already supplied by a prior *blob* command. The frontend just needs to connect it.

```
'M' SP <mode> SP <dataref> SP <path> LF
```

Here usually <*dataref*> must be either a mark reference (*:*<*idnum*>) set by a prior *blob* command, or a full 40-byte SHA-1 of an existing Git blob object. If <*mode*> is *040000*` then <*dataref*> must be the full 40-byte SHA-1 of an existing Git tree object or a mark reference set with *--import-marks*.

Inline data format

The data content for the file has not been supplied yet. The frontend wants to supply it as part of this modify command.

```
'M' SP <mode> SP 'inline' SP <path> LF
data
```

See below for a detailed description of the *data* command.

In both formats <*mode*> is the type of file entry, specified in octal. Git only supports the following modes:

- *100644* or *644*: A normal (not-executable) file. The majority of files in most projects use this mode. If in doubt, this is what you want.
- *100755* or *755*: A normal, but executable, file.
- *120000*: A symlink, the content of the file will be the link target.
- *160000*: A gitlink, SHA-1 of the object refers to a commit in another repository. Git links can only be specified either by SHA or through a commit mark. They are used to implement submodules.
- *040000*: A subdirectory. Subdirectories can only be specified by SHA or through a tree mark set with *--import-marks*.

In both formats <*path*> is the complete path of the file to be added (if not already existing) or modified (if already existing).

A <*path*> can be written as unquoted bytes or a C-style quoted string.

When a <*path*> does not start with a double quote (*"*), it is an unquoted string and is parsed as literal bytes without any escape sequences. However, if the filename contains *LF* or starts with double quote, it cannot be represented as an unquoted string and must be quoted. Additionally, the source <*path*> in *filecopy* or *filerename* must be quoted if it contains SP.

When a <*path*> starts with a double quote (*"*), it is a C-style quoted string, where the complete filename is enclosed in a pair of double quotes and escape sequences are used. Certain characters must be escaped by preceding them with a backslash: *LF* is written as *\n*, backslash as **, and double quote as *\"*. Some characters may optionally be written with escape sequences: *\a* for bell, *\b* for backspace, *\f* for form feed, *\n* for line feed, *\r* for carriage return, *\t* for horizontal tab, and *\v* for vertical tab. Any byte can be written with 3-digit octal codes (e.g., *\033*). All filenames can be represented as quoted strings.

A <*path*> must use UNIX-style directory separators (forward slash */*) and its value must be in canonical form. That is it must not:

- contain an empty directory component (e.g. *foo//bar* is invalid),

- end with a directory separator (e.g. *foo/* is invalid),
- start with a directory separator (e.g. */foo* is invalid),
- contain the special component . or .. (e.g. *foo/./bar* and *foo/../bar* are invalid).

The root of the tree can be represented by an empty string as <*path*>.

<*path*> cannot contain NUL, either literally or escaped as *\000*. It is recommended that <*path*> always be encoded using UTF-8.

4.7. *filedelete*

Included in a *commit* command to remove a file or recursively delete an entire directory from the branch. If the file or directory removal makes its parent directory empty, the parent directory will be automatically removed too. This cascades up the tree until the first non-empty directory or the root is reached.

```
'D' SP <path> LF
```

here <*path*> is the complete path of the file or subdirectory to be removed from the branch. See *filemodify* above for a detailed description of <*path*>.

4.8. *filecopy*

Recursively copies an existing file or subdirectory to a different location within the branch. The existing file or directory must exist. If the destination exists it will be completely replaced by the content copied from the source.

```
'C' SP <path> SP <path> LF
```

here the first <*path*> is the source location and the second <*path*> is the destination. See *filemodify* above for a detailed description of what <*path*> may look like. To use a source path that contains SP the path must be quoted.

A *filecopy* command takes effect immediately. Once the source location has been copied to the destination any future commands applied to the source location will not impact the destination of the copy.

4.9. *filerename*

Renames an existing file or subdirectory to a different location within the branch. The existing file or directory must exist. If the destination exists it will be replaced by the source directory.

```
'R' SP <path> SP <path> LF
```

here the first <*path*> is the source location and the second <*path*> is the destination. See *filemodify* above for a detailed description of what <*path*> may look like. To use a source path that contains SP the path must be quoted.

A *filerename* command takes effect immediately. Once the source location has been renamed to the destination any future commands applied to the source location will create new files there and not impact the destination of the rename.

Note that a *filerename* is the same as a *filecopy* followed by a *filedelete* of the source location. There is a slight performance advantage to using *filerename*, but the advantage is so small that it is never worth trying to convert a delete/add pair in source material into a rename for fast-import. This *filerename* command is provided just to simplify frontends that already have rename information and don't want bother with decomposing it into a *filecopy* followed by a *filedelete*.

4.10. *filedeleteall*

Included in a *commit* command to remove all files (and also all directories) from the branch. This command resets the internal branch structure to have no files in it, allowing the frontend to subsequently add all interesting files from scratch.

```
'deleteall' LF
```

This command is extremely useful if the frontend does not know (or does not care to know) what files are currently on the branch, and therefore cannot generate the proper *filedelete* commands to update the content.

Issuing a *filedeleteall* followed by the needed *filemodify* commands to set the correct content will produce the same results as sending only the needed *filemodify* and *filedelete* commands. The *filedeleteall* approach may however require fast-import to use slightly more memory per active branch (less than 1 MiB for even most large projects); so frontends that can easily obtain only the affected paths for a commit are encouraged to do so.

4.11. *notemodify*

Included in a *commit* <*notes-ref*> command to add a new note annotating a <*commit-ish*> or change this annotation contents. Internally it is similar to filemodify 100644 on <*commit-ish*> path (maybe split into subdirectories). It's not advised to use any other commands to write to the <*notes-ref*> tree except *filedeleteall* to delete all existing notes in this tree. This command has two different means of specifying the content of the note.

External data format

The data content for the note was already supplied by a prior *blob* command. The frontend just needs to connect it to the commit that is to be annotated.

```
'N' SP <dataref> SP <commit-ish> LF
```

Here <*dataref*> can be either a mark reference (*:*<*idnum*>) set by a prior *blob* command, or a full 40-byte SHA-1 of an existing Git blob object.

Inline data format

The data content for the note has not been supplied yet. The frontend wants to supply it as part of this modify command.

```
'N' SP 'inline' SP <commit-ish> LF
data
```

See below for a detailed description of the *data* command.

In both formats <*commit-ish*> is any of the commit specification expressions also accepted by *from* (see above).

5. *mark*

Arranges for fast-import to save a reference to the current object, allowing the frontend to recall this object at a future point in time, without knowing its SHA-1. Here the current object is the object creation command the *mark* command appears within. This can be *commit*, *tag*, and *blob*, but *commit* is the most common usage.

```
'mark' SP ':' <idnum> LF
```

where <*idnum*> is the number assigned by the frontend to this mark. The value of <*idnum*> is expressed as an ASCII decimal integer. The value 0 is reserved and cannot be used as a mark. Only values greater than or equal to 1 may be used as marks.

New marks are created automatically. Existing marks can be moved to another object simply by reusing the same <*idnum*> in another *mark* command.

6. *original-oid*

Provides the name of the object in the original source control system. fast-import will simply ignore this directive, but filter processes which operate on and modify the stream before feeding to fast-import may have uses for this information

```
'original-oid' SP <object-identifier> LF
```

where <*object-identifier*> is any string not containing LF.

7. *tag*

Creates an annotated tag referring to a specific commit. To create lightweight (non-annotated) tags see the *reset* command below.

```
'tag' SP <name> LF
mark?
'from' SP <commit-ish> LF
original-oid?
'tagger' (SP <name>)? SP LT <email> GT SP <when> LF
data
```

where <*name*> is the name of the tag to create.

Tag names are automatically prefixed with *refs/tags/* when stored in Git, so importing the CVS branch symbol *RELENG-1_0-FINAL* would use just *RELENG-1_0-FINAL* for <*name*>, and fast-import will write the corresponding ref as *refs/tags/RELENG-1_0-FINAL*.

The value of <*name*> must be a valid refname in Git and therefore may contain forward slashes. As *LF* is not valid in a Git refname, no quoting or escaping syntax is supported here.

The *from* command is the same as in the *commit* command; see above for details.

The *tagger* command uses the same format as *committer* within *commit*; again see above for details.

The *data* command following *tagger* must supply the annotated tag message (see below for *data* command syntax). To import an empty tag message use a 0 length data. Tag messages are free-form and are not interpreted by Git. Currently they must be encoded in UTF-8, as fast-import does not permit other encodings to be specified.

Signing annotated tags during import from within fast-import is not supported. Trying to include your own PGP/GPG signature is not recommended, as the frontend does not (easily) have access to the complete set of bytes which normally goes into such a signature. If signing is required, create lightweight tags from within fast-import with *reset*, then create the annotated versions of those tags offline with the standard *git tag* process.

8. *reset*

Creates (or recreates) the named branch, optionally starting from a specific revision. The reset command allows a frontend to issue a new *from* command for an existing branch, or to create a new branch from an existing commit without creating a new commit.

```
'reset' SP <ref> LF
('from' SP <commit-ish> LF)?
LF?
```

For a detailed description of <*ref*> and <*commit-ish*> see above under *commit* and *from*.

The *LF* after the command is optional (it used to be required).

The *reset* command can also be used to create lightweight (non-annotated) tags. For example:

```
reset refs/tags/938
from :938
```

would create the lightweight tag *refs/tags/938* referring to whatever commit mark *:938* references.

9. *blob*

Requests writing one file revision to the packfile. The revision is not connected to any commit; this connection must be formed in a subsequent *commit* command by referencing the blob through an assigned mark.

```
'blob' LF
mark?
original-oid?
data
```

The mark command is optional here as some frontends have chosen to generate the Git SHA-1 for the blob on their own, and feed that directly to *commit*. This is typically more work than it's worth however, as marks are inexpensive to store and easy to use.

10. *data*

Supplies raw data (for use as blob/file content, commit messages, or annotated tag messages) to fast-import. Data can be supplied using an exact byte count or delimited with a terminating line. Real frontends intended for production-quality conversions should always use the exact byte count format, as it is more robust and performs better. The delimited format is intended primarily for testing fast-import.

Comment lines appearing within the <*raw*> part of *data* commands are always taken to be part of the body of the data and are therefore never ignored by fast-import. This makes it safe to import any file/message content whose lines might start with *#*.

Exact byte count format

The frontend must specify the number of bytes of data.

```
'data' SP <count> LF
<raw> LF?
```

where <*count*> is the exact number of bytes appearing within <*raw*>. The value of <*count*> is expressed as an ASCII decimal integer. The *LF* on either side of <*raw*> is not included in <*count*> and will not be included in the imported data.

The *LF* after <*raw*> is optional (it used to be required) but recommended. Always including it makes debugging a fast-import stream easier as the next command always starts in column 0 of the next line, even if <*raw*> did not end with an *LF*.

Delimited format

A delimiter string is used to mark the end of the data. fast-import will compute the length by searching for the delimiter. This format is primarily useful for testing and is not recommended for real data.

```
'data' SP '<<' <delim> LF
<raw> LF
<delim> LF
LF?
```

where <*delim*> is the chosen delimiter string. The string <*delim*> must not appear on a line by itself within <*raw*>, as otherwise fast-import will think the data ends earlier than it really does. The *LF* immediately trailing <*raw*> is part of <*raw*>. This is one of the limitations of the delimited format, it is impossible to supply a data chunk which does not have an LF as its last byte.

The *LF* after <*delim*> *LF* is optional (it used to be required).

11. *alias*

Record that a mark refers to a given object without first creating any new object.

```
'alias' LF
mark
'to' SP <commit-ish> LF
LF?
```

For a detailed description of *<commit-ish>* see above under *from*.

12. *checkpoint*

Forces fast-import to close the current packfile, start a new one, and to save out all current branch refs, tags and marks.

```
'checkpoint' LF
LF?
```

Note that fast-import automatically switches packfiles when the current packfile reaches --max-pack-size, or 4 GiB, whichever limit is smaller. During an automatic packfile switch fast-import does not update the branch refs, tags or marks.

As a *checkpoint* can require a significant amount of CPU time and disk IO (to compute the overall pack SHA-1 checksum, generate the corresponding index file, and update the refs) it can easily take several minutes for a single *checkpoint* command to complete.

Frontends may choose to issue checkpoints during extremely large and long running imports, or when they need to allow another Git process access to a branch. However given that a 30 GiB Subversion repository can be loaded into Git through fast-import in about 3 hours, explicit checkpointing may not be necessary.

The *LF* after the command is optional (it used to be required).

13. *progress*

Causes fast-import to print the entire *progress* line unmodified to its standard output channel (file descriptor 1) when the command is processed from the input stream. The command otherwise has no impact on the current import, or on any of fast-import's internal state.

```
'progress' SP <any> LF
LF?
```

The *<any>* part of the command may contain any sequence of bytes that does not contain *LF*. The *LF* after the command is optional. Callers may wish to process the output through a tool such as sed to remove the leading part of the line, for example:

```
frontend | git fast-import | sed 's/^progress //'
```

Placing a *progress* command immediately after a *checkpoint* will inform the reader when the *checkpoint* has been completed and it can safely access the refs that fast-import updated.

14. *get-mark*

Causes fast-import to print the SHA-1 corresponding to a mark to stdout or to the file descriptor previously arranged with the *--cat-blob-fd* argument. The command otherwise has no impact on the current import; its purpose is to retrieve SHA-1s that later commits might want to refer to in their commit messages.

```
'get-mark' SP ':' <idnum> LF
```

See Responses To Commands below for details about how to read this output safely.

15. *cat-blob*

Causes fast-import to print a blob to a file descriptor previously arranged with the *--cat-blob-fd* argument. The command otherwise has no impact on the current import; its main purpose is to retrieve blobs that may be in fast-import's memory but not accessible from the target repository.

```
'cat-blob' SP <dataref> LF
```

The <*dataref*> can be either a mark reference (*:*<*idnum*>) set previously or a full 40-byte SHA-1 of a Git blob, preexisting or ready to be written.

Output uses the same format as *git cat-file --batch*:

```
<sha1> SP 'blob' SP <size> LF
<contents> LF
```

This command can be used where a *filemodify* directive can appear, allowing it to be used in the middle of a commit. For a *filemodify* using an inline directive, it can also appear right before the *data* directive.

See Responses To Commands below for details about how to read this output safely.

16. *ls*

Prints information about the object at a path to a file descriptor previously arranged with the *--cat-blob-fd* argument. This allows printing a blob from the active commit (with *cat-blob*) or copying a blob or tree from a previous commit for use in the current one (with *filemodify*).

The *ls* command can also be used where a *filemodify* directive can appear, allowing it to be used in the middle of a commit.

Reading from the active commit

> This form can only be used in the middle of a *commit*. The path names a directory entry within fast-import's active commit. The path must be quoted in this case.
>
> ```
> 'ls' SP <path> LF
> ```

Reading from a named tree

> The <*dataref*> can be a mark reference (*:*<*idnum*>) or the full 40-byte SHA-1 of a Git tag, commit, or tree object, preexisting or waiting to be written. The path is relative to the top level of the tree named by <*dataref*>.
>
> ```
> 'ls' SP <dataref> SP <path> LF
> ```

See *filemodify* above for a detailed description of <*path*>.

Output uses the same format as *git ls-tree* <*tree*> -- <*path*>:

```
<mode> SP ('blob' | 'tree' | 'commit') SP <dataref> HT <path> LF
```

The <dataref> represents the blob, tree, or commit object at <path> and can be used in later *get-mark*, *cat-blob*, *filemodify*, or *ls* commands.

If there is no file or subtree at that path, *git fast-import* will instead report

```
missing SP <path> LF
```

See Responses To Commands below for details about how to read this output safely.

17. *feature*

Require that fast-import supports the specified feature, or abort if it does not.

```
'feature' SP <feature> ('=' <argument>)? LF
```

The <feature> part of the command may be any one of the following:

date-format , export-marks , relative-marks , no-relative-marks , force

> Act as though the corresponding command-line option with a leading -- was passed on the command line (see OPTIONS, above).

import-marks , import-marks-if-exists

Like --import-marks except in two respects: first, only one "feature import-marks" or "feature import-marks-if-exists" command is allowed per stream; second, an --import-marks= or --import-marks-if-exists command-line option overrides any of these "feature" commands in the stream; third, "feature import-marks-if-exists" like a corresponding command-line option silently skips a nonexistent file.

get-mark , cat-blob , ls

Require that the backend support the *get-mark*, *cat-blob*, or *ls* command respectively. Versions of fast-import not supporting the specified command will exit with a message indicating so. This lets the import error out early with a clear message, rather than wasting time on the early part of an import before the unsupported command is detected.

notes

Require that the backend support the *notemodify* (N) subcommand to the *commit* command. Versions of fast-import not supporting notes will exit with a message indicating so.

done

Error out if the stream ends without a *done* command. Without this feature, errors causing the frontend to end abruptly at a convenient point in the stream can go undetected. This may occur, for example, if an import front end dies in mid-operation without emitting SIGTERM or SIGKILL at its subordinate git fast-import instance.

18. *option*

Processes the specified option so that git fast-import behaves in a way that suits the frontend's needs. Note that options specified by the frontend are overridden by any options the user may specify to git fast-import itself.

```
'option' SP <option> LF
```

The <*option*> part of the command may contain any of the options listed in the OPTIONS section that do not change import semantics, without the leading -- and is treated in the same way.

Option commands must be the first commands on the input (not counting feature commands), to give an option command after any non-option command is an error.

The following command-line options change import semantics and may therefore not be passed as option:

- date-format
- import-marks
- export-marks
- cat-blob-fd
- force

19. *done*

If the *done* feature is not in use, treated as if EOF was read. This can be used to tell fast-import to finish early.

If the *--done* command-line option or *feature done* command is in use, the *done* command is mandatory and marks the end of the stream.

RESPONSES TO COMMANDS

New objects written by fast-import are not available immediately. Most fast-import commands have no visible effect until the next checkpoint (or completion). The frontend can send commands to fill fast-import's input pipe without worrying about how quickly they will take effect, which improves performance by simplifying scheduling.

For some frontends, though, it is useful to be able to read back data from the current repository as it is being updated (for example when the source material describes objects in terms of patches to be applied to previously imported objects). This can be accomplished by connecting the frontend and fast-import via bidirectional pipes:

```
mkfifo fast-import-output
frontend <fast-import-output |
git fast-import >fast-import-output
```

A frontend set up this way can use *progress*, *get-mark*, *ls*, and *cat-blob* commands to read information from the import in progress.

To avoid deadlock, such frontends must completely consume any pending output from *progress*, *ls*, *get-mark*, and *cat-blob* before performing writes to fast-import that might block.

CRASH REPORTS

If fast-import is supplied invalid input it will terminate with a non-zero exit status and create a crash report in the top level of the Git repository it was importing into. Crash reports contain a snapshot of the internal fast-import state as well as the most recent commands that lead up to the crash.

All recent commands (including stream comments, file changes and progress commands) are shown in the command history within the crash report, but raw file data and commit messages are excluded from the crash report. This exclusion saves space within the report file and reduces the amount of buffering that fast-import must perform during execution.

After writing a crash report fast-import will close the current packfile and export the marks table. This allows the frontend developer to inspect the repository state and resume the import from the point where it crashed. The modified branches and tags are not updated during a crash, as the import did not complete successfully. Branch and tag information can be found in the crash report and must be applied manually if the update is needed.

An example crash:

```
$ cat >in <<END_OF_INPUT
# my very first test commit
commit refs/heads/master
committer Shawn O. Pearce <spearce> 19283 -0400
# who is that guy anyway?
data <<EOF
this is my commit
EOF
M 644 inline .gitignore
data <<EOF
.gitignore
EOF
M 777 inline bob
END_OF_INPUT

$ git fast-import <in
fatal: Corrupt mode: M 777 inline bob
fast-import: dumping crash report to .git/fast_import_crash_8434

$ cat .git/fast_import_crash_8434
fast-import crash report:
    fast-import process: 8434
    parent process     : 1391
    at Sat Sep 1 00:58:12 2007

fatal: Corrupt mode: M 777 inline bob

Most Recent Commands Before Crash
```

```
---------------------------------
  # my very first test commit
  commit refs/heads/master
  committer Shawn O. Pearce <spearce> 19283 -0400
  # who is that guy anyway?
  data <<EOF
  M 644 inline .gitignore
  data <<EOF
* M 777 inline bob

Active Branch LRU
-----------------
    active_branches = 1 cur, 5 max

pos  clock name
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 1)      0 refs/heads/master

Inactive Branches
-----------------
refs/heads/master:
  status      : active loaded dirty
  tip commit  : 0000000000000000000000000000000000000000
  old tree    : 0000000000000000000000000000000000000000
  cur tree    : 0000000000000000000000000000000000000000
  commit clock: 0
  last pack   :

-------------------
END OF CRASH REPORT
```

TIPS AND TRICKS

The following tips and tricks have been collected from various users of fast-import, and are offered here as suggestions.

1. Use One Mark Per Commit

When doing a repository conversion, use a unique mark per commit (*mark :<n>*) and supply the --export-marks option on the command line. fast-import will dump a file which lists every mark and the Git object SHA-1 that corresponds to it. If the frontend can tie the marks back to the source repository, it is easy to verify the accuracy and completeness of the import by comparing each Git commit to the corresponding source revision.

Coming from a system such as Perforce or Subversion, this should be quite simple, as the fast-import mark can also be the Perforce changeset number or the Subversion revision number.

2. Freely Skip Around Branches

Don't bother trying to optimize the frontend to stick to one branch at a time during an import. Although doing so might be slightly faster for fast-import, it tends to increase the complexity of the frontend code considerably.

The branch LRU builtin to fast-import tends to behave very well, and the cost of activating an inactive branch is so low that bouncing around between branches has virtually no impact on import performance.

3. Handling Renames

When importing a renamed file or directory, simply delete the old name(s) and modify the new name(s) during the corresponding commit. Git performs rename detection after-the-fact, rather than explicitly during a commit.

4. Use Tag Fixup Branches

Some other SCM systems let the user create a tag from multiple files which are not from the same commit/change-set. Or to create tags which are a subset of the files available in the repository.

Importing these tags as-is in Git is impossible without making at least one commit which fixes up the files to match the content of the tag. Use fast-import's *reset* command to reset a dummy branch outside of your normal branch space to the base commit for the tag, then commit one or more file fixup commits, and finally tag the dummy branch.

For example since all normal branches are stored under *refs/heads/* name the tag fixup branch *TAG_FIXUP*. This way it is impossible for the fixup branch used by the importer to have namespace conflicts with real branches imported from the source (the name *TAG_FIXUP* is not *refs/heads/TAG_FIXUP*).

When committing fixups, consider using *merge* to connect the commit(s) which are supplying file revisions to the fixup branch. Doing so will allow tools such as *git blame* to track through the real commit history and properly annotate the source files.

After fast-import terminates the frontend will need to do *rm .git/TAG_FIXUP* to remove the dummy branch.

5. Import Now, Repack Later

As soon as fast-import completes the Git repository is completely valid and ready for use. Typically this takes only a very short time, even for considerably large projects (100,000+ commits).

However repacking the repository is necessary to improve data locality and access performance. It can also take hours on extremely large projects (especially if -f and a large --window parameter is used). Since repacking is safe to run alongside readers and writers, run the repack in the background and let it finish when it finishes. There is no reason to wait to explore your new Git project!

If you choose to wait for the repack, don't try to run benchmarks or performance tests until repacking is completed. fast-import outputs suboptimal packfiles that are simply never seen in real use situations.

6. Repacking Historical Data

If you are repacking very old imported data (e.g. older than the last year), consider expending some extra CPU time and supplying --window=50 (or higher) when you run *git repack*. This will take longer, but will also produce a smaller packfile. You only need to expend the effort once, and everyone using your project will benefit from the smaller repository.

7. Include Some Progress Messages

Every once in a while have your frontend emit a *progress* message to fast-import. The contents of the messages are entirely free-form, so one suggestion would be to output the current month and year each time the current commit date moves into the next month. Your users will feel better knowing how much of the data stream has been processed.

PACKFILE OPTIMIZATION

When packing a blob fast-import always attempts to deltify against the last blob written. Unless specifically arranged for by the frontend, this will probably not be a prior version of the same file, so the generated delta will not be the smallest possible. The resulting packfile will be compressed, but will not be optimal.

Frontends which have efficient access to all revisions of a single file (for example reading an RCS/CVS ,v file) can choose to supply all revisions of that file as a sequence of consecutive *blob* commands. This allows fast-import to deltify the different file revisions against each other, saving space in the final packfile. Marks can be used to later identify individual file revisions during a sequence of *commit* commands.

The packfile(s) created by fast-import do not encourage good disk access patterns. This is caused by fast-import writing the data in the order it is received on standard input, while Git typically organizes data within packfiles to

make the most recent (current tip) data appear before historical data. Git also clusters commits together, speeding up revision traversal through better cache locality.

For this reason it is strongly recommended that users repack the repository with *git repack -a -d* after fast-import completes, allowing Git to reorganize the packfiles for faster data access. If blob deltas are suboptimal (see above) then also adding the *-f* option to force recomputation of all deltas can significantly reduce the final packfile size (30-50% smaller can be quite typical).

Instead of running *git repack* you can also run *git gc --aggressive*, which will also optimize other things after an import (e.g. pack loose refs). As noted in the "AGGRESSIVE" section in Section G.3.58, “git-gc(1)” the *--aggressive* option will find new deltas with the *-f* option to Section G.3.113, “git-repack(1)”. For the reasons elaborated on above using *--aggressive* after a fast-import is one of the few cases where it's known to be worthwhile.

MEMORY UTILIZATION

There are a number of factors which affect how much memory fast-import requires to perform an import. Like critical sections of core Git, fast-import uses its own memory allocators to amortize any overheads associated with malloc. In practice fast-import tends to amortize any malloc overheads to 0, due to its use of large block allocations.

1. per object

fast-import maintains an in-memory structure for every object written in this execution. On a 32 bit system the structure is 32 bytes, on a 64 bit system the structure is 40 bytes (due to the larger pointer sizes). Objects in the table are not deallocated until fast-import terminates. Importing 2 million objects on a 32 bit system will require approximately 64 MiB of memory.

The object table is actually a hashtable keyed on the object name (the unique SHA-1). This storage configuration allows fast-import to reuse an existing or already written object and avoid writing duplicates to the output packfile. Duplicate blobs are surprisingly common in an import, typically due to branch merges in the source.

2. per mark

Marks are stored in a sparse array, using 1 pointer (4 bytes or 8 bytes, depending on pointer size) per mark. Although the array is sparse, frontends are still strongly encouraged to use marks between 1 and n, where n is the total number of marks required for this import.

3. per branch

Branches are classified as active and inactive. The memory usage of the two classes is significantly different.

Inactive branches are stored in a structure which uses 96 or 120 bytes (32 bit or 64 bit systems, respectively), plus the length of the branch name (typically under 200 bytes), per branch. fast-import will easily handle as many as 10,000 inactive branches in under 2 MiB of memory.

Active branches have the same overhead as inactive branches, but also contain copies of every tree that has been recently modified on that branch. If subtree *include* has not been modified since the branch became active, its contents will not be loaded into memory, but if subtree *src* has been modified by a commit since the branch became active, then its contents will be loaded in memory.

As active branches store metadata about the files contained on that branch, their in-memory storage size can grow to a considerable size (see below).

fast-import automatically moves active branches to inactive status based on a simple least-recently-used algorithm. The LRU chain is updated on each *commit* command. The maximum number of active branches can be increased or decreased on the command line with --active-branches=.

4. per active tree

Trees (aka directories) use just 12 bytes of memory on top of the memory required for their entries (see per active file below). The cost of a tree is virtually 0, as its overhead amortizes out over the individual file entries.

5. per active file entry

Files (and pointers to subtrees) within active trees require 52 or 64 bytes (32/64 bit platforms) per entry. To conserve space, file and tree names are pooled in a common string table, allowing the filename Makefile to use just 16 bytes (after including the string header overhead) no matter how many times it occurs within the project.

The active branch LRU, when coupled with the filename string pool and lazy loading of subtrees, allows fast-import to efficiently import projects with 2,000+ branches and 45,114+ files in a very limited memory footprint (less than 2.7 MiB per active branch).

SIGNALS

Sending **SIGUSR1** to the *git fast-import* process ends the current packfile early, simulating a *checkpoint* command. The impatient operator can use this facility to peek at the objects and refs from an import in progress, at the cost of some added running time and worse compression.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

fastimport.unpackLimit

If the number of objects imported by Section G.3.47, “git-fast-import(1)” is below this limit, then the objects will be unpacked into loose object files. However, if the number of imported objects equals or exceeds this limit, then the pack will be stored as a pack. Storing the pack from a fast-import can make the import operation complete faster, especially on slow filesystems. If not set, the value of *transfer.unpackLimit* is used instead.

SEE ALSO

Section G.3.46, “git-fast-export(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.48. git-fetch-pack(1)

2024-07-02
Revision History
2024-07-02

NAME

git-fetch-pack - Receive missing objects from another repository

SYNOPSIS

git fetch-pack [--all] [--quiet|-q] [--keep|-k] [--thin] [--include-tag]
[--upload-pack=<git-upload-pack>]
[--depth=<n>] [--no-progress]
[-v] <repository> [<refs>…]

DESCRIPTION

Usually you would want to use *git fetch*, which is a higher level wrapper of this command, instead.

Invokes *git-upload-pack* on a possibly remote repository and asks it to send objects missing from this repository, to update the named heads. The list of commits available locally is found out by scanning the local refs/ hierarchy and sent to *git-upload-pack* running on the other end.

This command degenerates to download everything to complete the asked refs from the remote side when the local side does not have a common ancestor commit.

OPTIONS

--all

Fetch all remote refs.

--stdin

Take the list of refs from stdin, one per line. If there are refs specified on the command line in addition to this option, then the refs from stdin are processed after those on the command line.

If *--stateless-rpc* is specified together with this option then the list of refs must be in packet format (pkt-line). Each ref must be in a separate packet, and the list must end with a flush packet.

-q , --quiet

Pass *-q* flag to *git unpack-objects*; this makes the cloning process less verbose.

-k , --keep

Do not invoke *git unpack-objects* on received data, but create a single packfile out of it instead, and store it in the object database. If provided twice then the pack is locked against repacking.

--thin

Fetch a "thin" pack, which records objects in deltified form based on objects not included in the pack to reduce network traffic.

--include-tag

If the remote side supports it, annotated tags objects will be downloaded on the same connection as the other objects if the object the tag references is downloaded. The caller must otherwise determine the tags this option made available.

--upload-pack=<git-upload-pack>

Use this to specify the path to *git-upload-pack* on the remote side, if it is not found on your $PATH. Installations of sshd ignores the user's environment setup scripts for login shells (e.g. .bash_profile) and your privately installed git may not be found on the system default $PATH. Another workaround suggested is to set up your $PATH in ".bashrc", but this flag is for people who do not want to pay the overhead for non-interactive shells by having a lean .bashrc file (they set most of the things up in .bash_profile).

--exec=<git-upload-pack>

Same as --upload-pack=<git-upload-pack>.

--depth=<n>

Limit fetching to ancestor-chains not longer than n. *git-upload-pack* treats the special depth 2147483647 as infinite even if there is an ancestor-chain that long.

--shallow-since=<date>

Deepen or shorten the history of a shallow repository to include all reachable commits after <date>.

--shallow-exclude=<revision>

Deepen or shorten the history of a shallow repository to exclude commits reachable from a specified remote branch or tag. This option can be specified multiple times.

--deepen-relative

Argument --depth specifies the number of commits from the current shallow boundary instead of from the tip of each remote branch history.

--refetch

Skips negotiating commits with the server in order to fetch all matching objects. Use to reapply a new partial clone blob/tree filter.

--no-progress

Do not show the progress.

--check-self-contained-and-connected

Output "connectivity-ok" if the received pack is self-contained and connected.

-v

Run verbosely.

<repository>

The URL to the remote repository.

<refs>…

The remote heads to update from. This is relative to $GIT_DIR (e.g. "HEAD", "refs/heads/master"). When unspecified, update from all heads the remote side has.

If the remote has enabled the options *uploadpack.allowTipSHA1InWant*, *uploadpack.allowReachableSHA1InWant*, or *uploadpack.allowAnySHA1InWant*, they may alternatively be 40-hex sha1s present on the remote.

SEE ALSO

Section G.3.49, “git-fetch(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.49. git-fetch(1)

2024-07-02

Revision History

2024-07-02

NAME

git-fetch - Download objects and refs from another repository

SYNOPSIS

git fetch [<options>] [<repository> [<refspec>…]]
git fetch [<options>] <group>
git fetch --multiple [<options>] [(<repository> | <group>)…]
git fetch --all [<options>]

DESCRIPTION

Fetch branches and/or tags (collectively, "refs") from one or more other repositories, along with the objects necessary to complete their histories. Remote-tracking branches are updated (see the description of <refspec> below for ways to control this behavior).

By default, any tag that points into the histories being fetched is also fetched; the effect is to fetch tags that point at branches that you are interested in. This default behavior can be changed by using the --tags or --no-tags options

or by configuring remote.<name>.tagOpt. By using a refspec that fetches tags explicitly, you can fetch tags that do not point into branches you are interested in as well.

git fetch can fetch from either a single named repository or URL, or from several repositories at once if <group> is given and there is a remotes.<group> entry in the configuration file. (See Section G.3.29, “git-config(1)”).

When no remote is specified, by default the *origin* remote will be used, unless there's an upstream branch configured for the current branch.

The names of refs that are fetched, together with the object names they point at, are written to *.git/FETCH_HEAD*. This information may be used by scripts or other git commands, such as Section G.3.102, “git-pull(1)”.

OPTIONS

--[no-]all

Fetch all remotes. This overrides the configuration variable *fetch.all*.

-a , --append

Append ref names and object names of fetched refs to the existing contents of *.git/FETCH_HEAD*. Without this option old data in *.git/FETCH_HEAD* will be overwritten.

--atomic

Use an atomic transaction to update local refs. Either all refs are updated, or on error, no refs are updated.

--depth=<depth>

Limit fetching to the specified number of commits from the tip of each remote branch history. If fetching to a *shallow* repository created by *git clone* with *--depth=<depth>* option (see Section G.3.24, “git-clone(1)”), deepen or shorten the history to the specified number of commits. Tags for the deepened commits are not fetched.

--deepen=<depth>

Similar to --depth, except it specifies the number of commits from the current shallow boundary instead of from the tip of each remote branch history.

--shallow-since=<date>

Deepen or shorten the history of a shallow repository to include all reachable commits after <date>.

--shallow-exclude=<revision>

Deepen or shorten the history of a shallow repository to exclude commits reachable from a specified remote branch or tag. This option can be specified multiple times.

--unshallow

If the source repository is complete, convert a shallow repository to a complete one, removing all the limitations imposed by shallow repositories.

If the source repository is shallow, fetch as much as possible so that the current repository has the same history as the source repository.

--update-shallow

By default when fetching from a shallow repository, *git fetch* refuses refs that require updating .git/shallow. This option updates .git/shallow and accepts such refs.

--negotiation-tip=<commit|glob>

By default, Git will report, to the server, commits reachable from all local refs to find common commits in an attempt to reduce the size of the to-be-received packfile. If specified, Git will only report commits reachable from the given tips. This is useful to speed up fetches when the user knows which local ref is likely to have commits in common with the upstream ref being fetched.

This option may be specified more than once; if so, Git will report commits reachable from any of the given commits.

The argument to this option may be a glob on ref names, a ref, or the (possibly abbreviated) SHA-1 of a commit. Specifying a glob is equivalent to specifying this option multiple times, one for each matching ref name.

See also the *fetch.negotiationAlgorithm* and *push.negotiate* configuration variables documented in Section G.3.29, "git-config(1)", and the *--negotiate-only* option below.

--negotiate-only

Do not fetch anything from the server, and instead print the ancestors of the provided *--negotiation-tip=** arguments, which we have in common with the server.

This is incompatible with *--recurse-submodules=[yes|on-demand]*. Internally this is used to implement the *push.negotiate* option, see Section G.3.29, "git-config(1)".

--dry-run

Show what would be done, without making any changes.

--porcelain

Print the output to standard output in an easy-to-parse format for scripts. See section OUTPUT in Section G.3.49, "git-fetch(1)" for details.

This is incompatible with *--recurse-submodules=[yes|on-demand]* and takes precedence over the *fetch.output* config option.

--[no-]write-fetch-head

Write the list of remote refs fetched in the *FETCH_HEAD* file directly under *$GIT_DIR*. This is the default. Passing *--no-write-fetch-head* from the command line tells Git not to write the file. Under *--dry-run* option, the file is never written.

-f , --force

When *git fetch* is used with *<src>:<dst>* refspec, it may refuse to update the local branch as discussed in the *<refspec>* part below. This option overrides that check.

-k , --keep

Keep downloaded pack.

--multiple

Allow several <repository> and <group> arguments to be specified. No <refspec>s may be specified.

--[no-]auto-maintenance , --[no-]auto-gc

Run *git maintenance run --auto* at the end to perform automatic repository maintenance if needed. (*--[no-]auto-gc* is a synonym.) This is enabled by default.

--[no-]write-commit-graph

Write a commit-graph after fetching. This overrides the config setting *fetch.writeCommitGraph*.

--prefetch

Modify the configured refspec to place all refs into the *refs/prefetch/* namespace. See the *prefetch* task in Section G.3.80, “git-maintenance(1)”.

-p , --prune

Before fetching, remove any remote-tracking references that no longer exist on the remote. Tags are not subject to pruning if they are fetched only because of the default tag auto-following or due to a --tags option. However, if tags are fetched due to an explicit refspec (either on the command line or in the remote configuration, for example if the remote was cloned with the --mirror option), then they are also subject to pruning. Supplying *--prune-tags* is a shorthand for providing the tag refspec.

See the PRUNING section below for more details.

-P , --prune-tags

Before fetching, remove any local tags that no longer exist on the remote if *--prune* is enabled. This option should be used more carefully, unlike *--prune* it will remove any local references (local tags) that have been created. This option is a shorthand for providing the explicit tag refspec along with *--prune*, see the discussion about that in its documentation.

See the PRUNING section below for more details.

-n , --no-tags

By default, tags that point at objects that are downloaded from the remote repository are fetched and stored locally. This option disables this automatic tag following. The default behavior for a remote may be specified with the remote.<name>.tagOpt setting. See Section G.3.29, “git-config(1)”.

--refetch

Instead of negotiating with the server to avoid transferring commits and associated objects that are already present locally, this option fetches all objects as a fresh clone would. Use this to reapply a partial clone filter from configuration or using *--filter=* when the filter definition has changed. Automatic post-fetch maintenance will perform object database pack consolidation to remove any duplicate objects.

--refmap=<refspec>

When fetching refs listed on the command line, use the specified refspec (can be given more than once) to map the refs to remote-tracking branches, instead of the values of *remote.*.fetch* configuration variables for the remote repository. Providing an empty *<refspec>* to the *--refmap* option causes Git to ignore the configured refspecs and rely entirely on the refspecs supplied as command-line arguments. See section on "Configured Remote-tracking Branches" for details.

-t , --tags

Fetch all tags from the remote (i.e., fetch remote tags *refs/tags/** into local tags with the same name), in addition to whatever else would otherwise be fetched. Using this option alone does not subject tags to pruning, even if --prune is used (though tags may be pruned anyway if they are also the destination of an explicit refspec; see *--prune*).

--recurse-submodules[=(yes|on-demand|no)]

This option controls if and under what conditions new commits of submodules should be fetched too. When recursing through submodules, *git fetch* always attempts to fetch "changed" submodules, that is, a submodule that has commits that are referenced by a newly fetched superproject commit but are missing in the local submodule clone. A changed submodule can be fetched as long as it is present locally e.g. in *$GIT_DIR/*

modules/ (see Section G.4.15, “gitsubmodules(7)”); if the upstream adds a new submodule, that submodule cannot be fetched until it is cloned e.g. by *git submodule update*.

When set to *on-demand*, only changed submodules are fetched. When set to *yes*, all populated submodules are fetched and submodules that are both unpopulated and changed are fetched. When set to *no*, submodules are never fetched.

When unspecified, this uses the value of *fetch.recurseSubmodules* if it is set (see Section G.3.29, “git-config(1)”), defaulting to *on-demand* if unset. When this option is used without any value, it defaults to *yes*.

-j , --jobs=<n>

Number of parallel children to be used for all forms of fetching.

If the *--multiple* option was specified, the different remotes will be fetched in parallel. If multiple submodules are fetched, they will be fetched in parallel. To control them independently, use the config settings *fetch.parallel* and *submodule.fetchJobs* (see Section G.3.29, “git-config(1)”).

Typically, parallel recursive and multi-remote fetches will be faster. By default fetches are performed sequentially, not in parallel.

--no-recurse-submodules

Disable recursive fetching of submodules (this has the same effect as using the *--recurse-submodules=no* option).

--set-upstream

If the remote is fetched successfully, add upstream (tracking) reference, used by argument-less Section G.3.102, “git-pull(1)” and other commands. For more information, see *branch.<name>.merge* and *branch.<name>.remote* in Section G.3.29, “git-config(1)”.

--submodule-prefix=<path>

Prepend <path> to paths printed in informative messages such as "Fetching submodule foo". This option is used internally when recursing over submodules.

--recurse-submodules-default=[yes|on-demand]

This option is used internally to temporarily provide a non-negative default value for the --recurse-submodules option. All other methods of configuring fetch's submodule recursion (such as settings in Section G.4.10, “gitmodules(5)” and Section G.3.29, “git-config(1)”) override this option, as does specifying --[no-]recurse-submodules directly.

-u , --update-head-ok

By default *git fetch* refuses to update the head which corresponds to the current branch. This flag disables the check. This is purely for the internal use for *git pull* to communicate with *git fetch*, and unless you are implementing your own Porcelain you are not supposed to use it.

--upload-pack <upload-pack>

When given, and the repository to fetch from is handled by *git fetch-pack*, *--exec=<upload-pack>* is passed to the command to specify non-default path for the command run on the other end.

-q , --quiet

Pass --quiet to git-fetch-pack and silence any other internally used git commands. Progress is not reported to the standard error stream.

-v , --verbose

Be verbose.

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

-o <option> , --server-option=<option>

Transmit the given string to the server when communicating using protocol version 2. The given string must not contain a NUL or LF character. The server's handling of server options, including unknown ones, is server-specific. When multiple *--server-option=<option>* are given, they are all sent to the other side in the order listed on the command line.

--show-forced-updates

By default, git checks if a branch is force-updated during fetch. This can be disabled through fetch.showForcedUpdates, but the --show-forced-updates option guarantees this check occurs. See Section G.3.29, "git-config(1)".

--no-show-forced-updates

By default, git checks if a branch is force-updated during fetch. Pass --no-show-forced-updates or set fetch.showForcedUpdates to false to skip this check for performance reasons. If used during *git-pull* the --ff-only option will still check for forced updates before attempting a fast-forward update. See Section G.3.29, "git-config(1)".

-4 , --ipv4

Use IPv4 addresses only, ignoring IPv6 addresses.

-6 , --ipv6

Use IPv6 addresses only, ignoring IPv4 addresses.

<repository>

The "remote" repository that is the source of a fetch or pull operation. This parameter can be either a URL (see the section GIT URLS below) or the name of a remote (see the section REMOTES below).

<group>

A name referring to a list of repositories as the value of remotes.<group> in the configuration file. (See Section G.3.29, "git-config(1)").

<refspec>

Specifies which refs to fetch and which local refs to update. When no <refspec>s appear on the command line, the refs to fetch are read from *remote.<repository>.fetch* variables instead (see CONFIGURED REMOTE-TRACKING BRANCHES below).

The format of a <refspec> parameter is an optional plus +, followed by the source <src>, followed by a colon :, followed by the destination ref <dst>. The colon can be omitted when <dst> is empty. <src> is typically a ref, but it can also be a fully spelled hex object name.

A <refspec> may contain a * in its <src> to indicate a simple pattern match. Such a refspec functions like a glob that matches any ref with the same prefix. A pattern <refspec> must have a * in both the <src> and <dst>. It will map refs to the destination by replacing the * with the contents matched from the source.

If a refspec is prefixed by ^, it will be interpreted as a negative refspec. Rather than specifying which refs to fetch or which local refs to update, such a refspec will instead specify refs to exclude. A ref will be considered to match if it matches at least one positive refspec, and does not match any negative refspec. Negative refspecs can be useful to restrict the scope of a pattern refspec so that it will not include specific refs. Negative refspecs can themselves be pattern refspecs. However, they may only contain a <src> and do not specify a <dst>. Fully spelled out hex object names are also not supported.

tag <*tag*> means the same as *refs/tags/*<*tag*>*:refs/tags/*<*tag*>; it requests fetching everything up to the given tag.

The remote ref that matches <src> is fetched, and if <dst> is not an empty string, an attempt is made to update the local ref that matches it.

Whether that update is allowed without *--force* depends on the ref namespace it's being fetched to, the type of object being fetched, and whether the update is considered to be a fast-forward. Generally, the same rules apply for fetching as when pushing, see the <*refspec*>... section of Section G.3.103, "git-push(1)" for what those are. Exceptions to those rules particular to *git fetch* are noted below.

Until Git version 2.20, and unlike when pushing with Section G.3.103, "git-push(1)", any updates to *refs/tags/** would be accepted without + in the refspec (or *--force*). When fetching, we promiscuously considered all tag updates from a remote to be forced fetches. Since Git version 2.20, fetching to update *refs/tags/** works the same way as when pushing. I.e. any updates will be rejected without + in the refspec (or *--force*).

Unlike when pushing with Section G.3.103, "git-push(1)", any updates outside of *refs/{tags,heads}/** will be accepted without + in the refspec (or *--force*), whether that's swapping e.g. a tree object for a blob, or a commit for another commit that doesn't have the previous commit as an ancestor etc.

Unlike when pushing with Section G.3.103, "git-push(1)", there is no configuration which'll amend these rules, and nothing like a *pre-fetch* hook analogous to the *pre-receive* hook.

As with pushing with Section G.3.103, "git-push(1)", all of the rules described above about what's not allowed as an update can be overridden by adding an optional leading + to a refspec (or using the *--force* command line option). The only exception to this is that no amount of forcing will make the *refs/heads/** namespace accept a non-commit object.

> **Note**
>
> When the remote branch you want to fetch is known to be rewound and rebased regularly, it is expected that its new tip will not be a descendant of its previous tip (as stored in your remote-tracking branch the last time you fetched). You would want to use the + sign to indicate non-fast-forward updates will be needed for such branches. There is no way to determine or declare that a branch will be made available in a repository with this behavior; the pulling user simply must know this is the expected usage pattern for a branch.

--stdin

Read refspecs, one per line, from stdin in addition to those provided as arguments. The "tag <name>" format is not supported.

GIT URLS

In general, URLs contain information about the transport protocol, the address of the remote server, and the path to the repository. Depending on the transport protocol, some of this information may be absent.

Git supports ssh, git, http, and https protocols (in addition, ftp and ftps can be used for fetching, but this is inefficient and deprecated; do not use them).

The native transport (i.e. git:// URL) does no authentication and should be used with caution on unsecured networks.

The following syntaxes may be used with them:

- `ssh://[`*<user>*`@]`*<host>*`[:`*<port>*`]/`*<path-to-git-repo>*
- `git://`*<host>*`[:`*<port>*`]/`*<path-to-git-repo>*
- `http[s]://`*<host>*`[:`*<port>*`]/`*<path-to-git-repo>*

- `ftp[s]://`*<host>*`[:`*<port>*`]/`*<path-to-git-repo>*

An alternative scp-like syntax may also be used with the ssh protocol:

- [*<user>*`@`]*<host>*`:/`*<path-to-git-repo>*

This syntax is only recognized if there are no slashes before the first colon. This helps differentiate a local path that contains a colon. For example the local path *foo:bar* could be specified as an absolute path or *./foo:bar* to avoid being misinterpreted as an ssh url.

The ssh and git protocols additionally support `~`*<username>* expansion:

- `ssh://`[*<user>*`@`]*<host>*`[:`*<port>*`]/~`*<user>*`/`*<path-to-git-repo>*
- `git://`*<host>*`[:`*<port>*`]/~`*<user>*`/`*<path-to-git-repo>*
- [*<user>*`@`]*<host>*`:~`*<user>*`/`*<path-to-git-repo>*

For local repositories, also supported by Git natively, the following syntaxes may be used:

- */path/to/repo.git/*
- `file:///path/to/repo.git/`

These two syntaxes are mostly equivalent, except when cloning, when the former implies *--local* option. See Section G.3.24, “git-clone(1)” for details.

git clone, *git fetch* and *git pull*, but not *git push*, will also accept a suitable bundle file. See Section G.3.12, “git-bundle(1)”.

When Git doesn't know how to handle a certain transport protocol, it attempts to use the *remote-<transport>* remote helper, if one exists. To explicitly request a remote helper, the following syntax may be used:

- *<transport>*::*<address>*

where *<address>* may be a path, a server and path, or an arbitrary URL-like string recognized by the specific remote helper being invoked. See Section G.4.12, “gitremote-helpers(7)” for details.

If there are a large number of similarly-named remote repositories and you want to use a different format for them (such that the URLs you use will be rewritten into URLs that work), you can create a configuration section of the form:

```
[url "<actual-url-base>"]
    insteadOf = <other-url-base>
```

For example, with this:

```
[url "git://git.host.xz/"]
        insteadOf = host.xz:/path/to/
        insteadOf = work:
```

a URL like "work:repo.git" or like "host.xz:/path/to/repo.git" will be rewritten in any context that takes a URL to be "git://git.host.xz/repo.git".

If you want to rewrite URLs for push only, you can create a configuration section of the form:

```
[url "<actual-url-base>"]
    pushInsteadOf = <other-url-base>
```

For example, with this:

```
[url "ssh://example.org/"]
        pushInsteadOf = git://example.org/
```

a URL like "git://example.org/path/to/repo.git" will be rewritten to "ssh://example.org/path/to/repo.git" for pushes, but pulls will still use the original URL.

REMOTES

The name of one of the following can be used instead of a URL as *<repository>* argument:

- a remote in the Git configuration file: *$GIT_DIR/config*,
- a file in the *$GIT_DIR/remotes* directory, or
- a file in the *$GIT_DIR/branches* directory.

All of these also allow you to omit the refspec from the command line because they each contain a refspec which git will use by default.

1. Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using Section G.3.112, “git-remote(1)”, Section G.3.29, “git-config(1)” or even by a manual edit to the *$GIT_DIR/config* file. The URL of this remote will be used to access the repository. The refspec of this remote will be used by default when you do not provide a refspec on the command line. The entry in the config file would appear like this:

```
[remote "<name>"]
        url = <URL>
        pushurl = <pushurl>
        push = <refspec>
        fetch = <refspec>
```

The *<pushurl>* is used for pushes only. It is optional and defaults to *<URL>*. Pushing to a remote affects all defined pushurls or all defined urls if no pushurls are defined. Fetch, however, will only fetch from the first defined url if multiple urls are defined.

2. Named file in *$GIT_DIR/remotes*

You can choose to provide the name of a file in *$GIT_DIR/remotes*. The URL in this file will be used to access the repository. The refspec in this file will be used as default when you do not provide a refspec on the command line. This file should have the following format:

```
URL: one of the above URL formats
Push: <refspec>
Pull: <refspec>
```

Push: lines are used by *git push* and *Pull:* lines are used by *git pull* and *git fetch*. Multiple *Push:* and *Pull:* lines may be specified for additional branch mappings.

3. Named file in *$GIT_DIR/branches*

You can choose to provide the name of a file in *$GIT_DIR/branches*. The URL in this file will be used to access the repository. This file should have the following format:

```
<URL>#<head>
```

<URL> is required; *#<head>* is optional.

Depending on the operation, git will use one of the following refspecs, if you don't provide one on the command line. *<branch>* is the name of this file in *$GIT_DIR/branches* and *<head>* defaults to *master*.

git fetch uses:

```
refs/heads/<head>:refs/heads/<branch>
```

git push uses:

```
HEAD:refs/heads/<head>
```

CONFIGURED REMOTE-TRACKING BRANCHES

You often interact with the same remote repository by regularly and repeatedly fetching from it. In order to keep track of the progress of such a remote repository, *git fetch* allows you to configure *remote.<repository>.fetch* configuration variables.

Typically such a variable may look like this:

```
[remote "origin"]
        fetch = +refs/heads/*:refs/remotes/origin/*
```

This configuration is used in two ways:

- When *git fetch* is run without specifying what branches and/or tags to fetch on the command line, e.g. *git fetch origin* or *git fetch*, *remote.<repository>.fetch* values are used as the refspecs--they specify which refs to fetch and which local refs to update. The example above will fetch all branches that exist in the *origin* (i.e. any ref that matches the left-hand side of the value, *refs/heads/**) and update the corresponding remote-tracking branches in the *refs/remotes/origin/** hierarchy.
- When *git fetch* is run with explicit branches and/or tags to fetch on the command line, e.g. *git fetch origin master*, the <refspec>s given on the command line determine what are to be fetched (e.g. *master* in the example, which is a short-hand for *master:*, which in turn means "fetch the *master* branch but I do not explicitly say what remote-tracking branch to update with it from the command line"), and the example command will fetch *only* the *master* branch. The *remote.<repository>.fetch* values determine which remote-tracking branch, if any, is updated. When used in this way, the *remote.<repository>.fetch* values do not have any effect in deciding *what* gets fetched (i.e. the values are not used as refspecs when the command-line lists refspecs); they are only used to decide *where* the refs that are fetched are stored by acting as a mapping.

The latter use of the *remote.<repository>.fetch* values can be overridden by giving the *--refmap=<refspec>* parameter(s) on the command line.

PRUNING

Git has a default disposition of keeping data unless it's explicitly thrown away; this extends to holding onto local references to branches on remotes that have themselves deleted those branches.

If left to accumulate, these stale references might make performance worse on big and busy repos that have a lot of branch churn, and e.g. make the output of commands like *git branch -a --contains <commit>* needlessly verbose, as well as impacting anything else that'll work with the complete set of known references.

These remote-tracking references can be deleted as a one-off with either of:

```
# While fetching
$ git fetch --prune <name>

# Only prune, don't fetch
$ git remote prune <name>
```

To prune references as part of your normal workflow without needing to remember to run that, set *fetch.prune* globally, or *remote.<name>.prune* per-remote in the config. See Section G.3.29, “git-config(1)”.

Here's where things get tricky and more specific. The pruning feature doesn't actually care about branches, instead it'll prune local ⟷ remote-references as a function of the refspec of the remote (see *<refspec>* and CONFIGURED REMOTE-TRACKING BRANCHES above).

Therefore if the refspec for the remote includes e.g. *refs/tags/*:refs/tags/**, or you manually run e.g. *git fetch --prune <name> "refs/tags/*:refs/tags/*"* it won't be stale remote tracking branches that are deleted, but any local tag that doesn't exist on the remote.

This might not be what you expect, i.e. you want to prune remote *<name>*, but also explicitly fetch tags from it, so when you fetch from it you delete all your local tags, most of which may not have come from the *<name>* remote in the first place.

So be careful when using this with a refspec like *refs/tags/*:refs/tags/**, or any other refspec which might map references from multiple remotes to the same local namespace.

Since keeping up-to-date with both branches and tags on the remote is a common use-case the *--prune-tags* option can be supplied along with *--prune* to prune local tags that don't exist on the remote, and force-update those tags that differ. Tag pruning can also be enabled with *fetch.pruneTags* or *remote.<name>.pruneTags* in the config. See Section G.3.29, “git-config(1)”.

The *--prune-tags* option is equivalent to having *refs/tags/*:refs/tags/** declared in the refspecs of the remote. This can lead to some seemingly strange interactions:

```
# These both fetch tags
$ git fetch --no-tags origin 'refs/tags/*:refs/tags/*'
$ git fetch --no-tags --prune-tags origin
```

The reason it doesn't error out when provided without *--prune* or its config versions is for flexibility of the configured versions, and to maintain a 1=1 mapping between what the command line flags do, and what the configuration versions do.

It's reasonable to e.g. configure *fetch.pruneTags=true* in *~/.gitconfig* to have tags pruned whenever *git fetch --prune* is run, without making every invocation of *git fetch* without *--prune* an error.

Pruning tags with *--prune-tags* also works when fetching a URL instead of a named remote. These will all prune tags not found on origin:

```
$ git fetch origin --prune --prune-tags
$ git fetch origin --prune 'refs/tags/*:refs/tags/*'
$ git fetch <url-of-origin> --prune --prune-tags
$ git fetch <url-of-origin> --prune 'refs/tags/*:refs/tags/*'
```

OUTPUT

The output of "git fetch" depends on the transport method used; this section describes the output when fetching over the Git protocol (either locally or via ssh) and Smart HTTP protocol.

The status of the fetch is output in tabular form, with each line representing the status of a single ref. Each line is of the form:

```
 <flag> <summary> <from> -> <to> [<reason>]
```

When using *--porcelain*, the output format is intended to be machine-parseable. In contrast to the human-readable output formats it thus prints to standard output instead of standard error. Each line is of the form:

```
<flag> <old-object-id> <new-object-id> <local-reference>
```

The status of up-to-date refs is shown only if the --verbose option is used.

In compact output mode, specified with configuration variable fetch.output, if either entire *<from>* or *<to>* is found in the other string, it will be substituted with * in the other string. For example, *master -> origin/master* becomes *master -> origin/**.

flag

A single character indicating the status of the ref:

(space)

for a successfully fetched fast-forward;

+

for a successful forced update;

-

for a successfully pruned ref;

t

for a successful tag update;

*

for a successfully fetched new ref;

!

for a ref that was rejected or failed to update; and

=

for a ref that was up to date and did not need fetching.

summary

For a successfully fetched ref, the summary shows the old and new values of the ref in a form suitable for using as an argument to *git log* (this is *<old>..<new>* in most cases, and *<old>...<new>* for forced non-fast-forward updates).

from

The name of the remote ref being fetched from, minus its *refs/<type>/* prefix. In the case of deletion, the name of the remote ref is "(none)".

to

The name of the local ref being updated, minus its *refs/<type>/* prefix.

reason

A human-readable explanation. In the case of successfully fetched refs, no explanation is needed. For a failed ref, the reason for failure is described.

EXAMPLES

- Update the remote-tracking branches:

  ```
  $ git fetch origin
  ```

  The above command copies all branches from the remote *refs/heads/* namespace and stores them to the local *refs/remotes/origin/* namespace, unless the *remote.<repository>.fetch* option is used to specify a non-default refspec.

- Using refspecs explicitly:

  ```
  $ git fetch origin +seen:seen maint:tmp
  ```

  This updates (or creates, as necessary) branches *seen* and *tmp* in the local repository by fetching from the branches (respectively) *seen* and *maint* from the remote repository.

  The *seen* branch will be updated even if it does not fast-forward, because it is prefixed with a plus sign; *tmp* will not be.

- Peek at a remote's branch, without configuring the remote in your local repository:

  ```
  $ git fetch git://git.kernel.org/pub/scm/git/git.git maint
  $ git log FETCH_HEAD
  ```

  The first command fetches the *maint* branch from the repository at *git://git.kernel.org/pub/scm/git/git.git* and the second command uses *FETCH_HEAD* to examine the branch with Section G.3.74, "git-log(1)". The fetched objects will eventually be removed by git's built-in housekeeping (see Section G.3.58, "git-gc(1)").

SECURITY

The fetch and push protocols are not designed to prevent one side from stealing data from the other repository that was not intended to be shared. If you have private data that you need to protect from a malicious peer, your best option is to store it in another repository. This applies to both clients and servers. In particular, namespaces on a server are not effective for read access control; you should only grant read access to a namespace to clients that you would trust with read access to the entire repository.

The known attack vectors are as follows:

1. The victim sends "have" lines advertising the IDs of objects it has that are not explicitly intended to be shared but can be used to optimize the transfer if the peer also has them. The attacker chooses an object ID X to steal and sends a ref to X, but isn't required to send the content of X because the victim already has it. Now the victim believes that the attacker has X, and it sends the content of X back to the attacker later. (This attack is most straightforward for a client to perform on a server, by creating a ref to X in the namespace the client has access to and then fetching it. The most likely way for a server to perform it on a client is to "merge" X into a public branch and hope that the user does additional work on this branch and pushes it back to the server without noticing the merge.)

2. As in #1, the attacker chooses an object ID X to steal. The victim sends an object Y that the attacker already has, and the attacker falsely claims to have X and not Y, so the victim sends Y as a delta against X. The delta reveals regions of X that are similar to Y to the attacker.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, "git-config(1)" documentation. The content is the same as what's found there:

fetch.recurseSubmodules

This option controls whether *git fetch* (and the underlying fetch in *git pull*) will recursively fetch into populated submodules. This option can be set either to a boolean value or to *on-demand*. Setting it to a boolean changes the behavior of fetch and pull to recurse unconditionally into submodules when set to true or to not recurse at all when set to false. When set to *on-demand*, fetch and pull will only recurse into a populated submodule when its superproject retrieves a commit that updates the submodule's reference. Defaults to *on-demand*, or to the value of *submodule.recurse* if set.

fetch.fsckObjects

If it is set to true, git-fetch-pack will check all fetched objects. See *transfer.fsckObjects* for what's checked. Defaults to false. If not set, the value of *transfer.fsckObjects* is used instead.

fetch.fsck.<msg-id>

Acts like *fsck.<msg-id>*, but is used by Section G.3.48, "git-fetch-pack(1)" instead of Section G.3.56, "git-fsck(1)". See the *fsck.<msg-id>* documentation for details.

fetch.fsck.skipList

Acts like *fsck.skipList*, but is used by Section G.3.48, "git-fetch-pack(1)" instead of Section G.3.56, "git-fsck(1)". See the *fsck.skipList* documentation for details.

fetch.unpackLimit

If the number of objects fetched over the Git native transfer is below this limit, then the objects will be unpacked into loose object files. However if the number of received objects equals or exceeds this limit then the received pack will be stored as a pack, after adding any missing delta bases. Storing the pack from a push can make the push operation complete faster, especially on slow filesystems. If not set, the value of *transfer.unpackLimit* is used instead.

fetch.prune

If true, fetch will automatically behave as if the *--prune* option was given on the command line. See also *remote.<name>.prune* and the PRUNING section of Section G.3.49, “git-fetch(1)”.

fetch.pruneTags

If true, fetch will automatically behave as if the *refs/tags/*:refs/tags/** refspec was provided when pruning, if not set already. This allows for setting both this option and *fetch.prune* to maintain a 1=1 mapping to upstream refs. See also *remote.<name>.pruneTags* and the PRUNING section of Section G.3.49, “git-fetch(1)”.

fetch.all

If true, fetch will attempt to update all available remotes. This behavior can be overridden by passing *--no-all* or by explicitly specifying one or more remote(s) to fetch from. Defaults to false.

fetch.output

Control how ref update status is printed. Valid values are *full* and *compact*. Default value is *full*. See the OUTPUT section in Section G.3.49, “git-fetch(1)” for details.

fetch.negotiationAlgorithm

Control how information about the commits in the local repository is sent when negotiating the contents of the packfile to be sent by the server. Set to "consecutive" to use an algorithm that walks over consecutive commits checking each one. Set to "skipping" to use an algorithm that skips commits in an effort to converge faster, but may result in a larger-than-necessary packfile; or set to "noop" to not send any information at all, which will almost certainly result in a larger-than-necessary packfile, but will skip the negotiation step. Set to "default" to override settings made previously and use the default behaviour. The default is normally "consecutive", but if *feature.experimental* is true, then the default is "skipping". Unknown values will cause *git fetch* to error out.

See also the *--negotiate-only* and *--negotiation-tip* options to Section G.3.49, “git-fetch(1)”.

fetch.showForcedUpdates

Set to false to enable *--no-show-forced-updates* in Section G.3.49, “git-fetch(1)” and Section G.3.102, “git-pull(1)” commands. Defaults to true.

fetch.parallel

Specifies the maximal number of fetch operations to be run in parallel at a time (submodules, or remotes when the *--multiple* option of Section G.3.49, “git-fetch(1)” is in effect).

A value of 0 will give some reasonable default. If unset, it defaults to 1.

For submodules, this setting can be overridden using the *submodule.fetchJobs* config setting.

fetch.writeCommitGraph

Set to true to write a commit-graph after every *git fetch* command that downloads a pack-file from a remote. Using the *--split* option, most executions will create a very small commit-graph file on top of the existing commit-graph file(s). Occasionally, these files will merge and the write may take longer. Having an updated commit-graph file helps performance of many Git commands, including *git merge-base*, *git push -f*, and *git log --graph*. Defaults to false.

fetch.bundleURI

This value stores a URI for downloading Git object data from a bundle URI before performing an incremental fetch from the origin Git server. This is similar to how the *--bundle-uri* option behaves in Section G.3.24, “git-clone(1)”. *git clone --bundle-uri* will set the *fetch.bundleURI* value if the supplied bundle URI contains a bundle list that is organized for incremental fetches.

If you modify this value and your repository has a *fetch.bundleCreationToken* value, then remove that *fetch.bundleCreationToken* value before fetching from the new bundle URI.

fetch.bundleCreationToken

When using *fetch.bundleURI* to fetch incrementally from a bundle list that uses the "creationToken" heuristic, this config value stores the maximum *creationToken* value of the downloaded bundles. This value is used to prevent downloading bundles in the future if the advertised *creationToken* is not strictly larger than this value.

The creation token values are chosen by the provider serving the specific bundle URI. If you modify the URI at *fetch.bundleURI*, then be sure to remove the value for the *fetch.bundleCreationToken* value before fetching.

BUGS

Using --recurse-submodules can only fetch new commits in submodules that are present locally e.g. in *$GIT_DIR/modules/*. If the upstream adds a new submodule, that submodule cannot be fetched until it is cloned e.g. by *git submodule update*. This is expected to be fixed in a future Git version.

SEE ALSO

Section G.3.102, “git-pull(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.50. git-filter-branch(1)

2024-07-02
Revision History
2024-07-02

NAME

git-filter-branch - Rewrite branches

SYNOPSIS

git filter-branch [--setup <command>] [--subdirectory-filter <directory>]
[--env-filter <command>] [--tree-filter <command>]
[--index-filter <command>] [--parent-filter <command>]
[--msg-filter <command>] [--commit-filter <command>]
[--tag-name-filter <command>] [--prune-empty]
[--original <namespace>] [-d <directory>] [-f | --force]
[--state-branch <branch>] [--] [<rev-list-options>…]

WARNING

git filter-branch has a plethora of pitfalls that can produce non-obvious manglings of the intended history rewrite (and can leave you with little time to investigate such problems since it has such abysmal performance). These safety and performance issues cannot be backward compatibly fixed and as such, its use is not recommended. Please use an alternative history filtering tool such as git filter-repo [https://github.com/newren/git-filter-repo/]. If you still need to use *git filter-branch*, please carefully read the section called “SAFETY” (and the section called “PERFORMANCE”) to learn about the land mines of filter-branch, and then vigilantly avoid as many of the hazards listed there as reasonably possible.

DESCRIPTION

Lets you rewrite Git revision history by rewriting the branches mentioned in the <rev-list-options>, applying custom filters on each revision. Those filters can modify each tree (e.g. removing a file or running a perl rewrite on all files) or information about each commit. Otherwise, all information (including original commit times or merge information) will be preserved.

The command will only rewrite the *positive* refs mentioned in the command line (e.g. if you pass *a..b*, only *b* will be rewritten). If you specify no filters, the commits will be recommitted without any changes, which would normally have no effect. Nevertheless, this may be useful in the future for compensating for some Git bugs or such, therefore such a usage is permitted.

NOTE: This command honors *.git/info/grafts* file and refs in the *refs/replace/* namespace. If you have any grafts or replacement refs defined, running this command will make them permanent.

WARNING! The rewritten history will have different object names for all the objects and will not converge with the original branch. You will not be able to easily push and distribute the rewritten branch on top of the original branch. Please do not use this command if you do not know the full implications, and avoid using it anyway, if a simple single commit would suffice to fix your problem. (See the "RECOVERING FROM UPSTREAM REBASE" section in Section G.3.107, “git-rebase(1)” for further information about rewriting published history.)

Always verify that the rewritten version is correct: The original refs, if different from the rewritten ones, will be stored in the namespace *refs/original/*.

Note that since this operation is very I/O expensive, it might be a good idea to redirect the temporary directory off-disk with the *-d* option, e.g. on tmpfs. Reportedly the speedup is very noticeable.

1. Filters

The filters are applied in the order as listed below. The <command> argument is always evaluated in the shell context using the *eval* command (with the notable exception of the commit filter, for technical reasons). Prior to that, the *$GIT_COMMIT* environment variable will be set to contain the id of the commit being rewritten. Also, GIT_AUTHOR_NAME, GIT_AUTHOR_EMAIL, GIT_AUTHOR_DATE, GIT_COMMITTER_NAME, GIT_COMMITTER_EMAIL, and GIT_COMMITTER_DATE are taken from the current commit and exported to the environment, in order to affect the author and committer identities of the replacement commit created by Section G.3.27, “git-commit-tree(1)” after the filters have run.

If any evaluation of <command> returns a non-zero exit status, the whole operation will be aborted.

A *map* function is available that takes an "original sha1 id" argument and outputs a "rewritten sha1 id" if the commit has been already rewritten, and "original sha1 id" otherwise; the *map* function can return several ids on separate lines if your commit filter emitted multiple commits.

OPTIONS

--setup <command>

This is not a real filter executed for each commit but a one time setup just before the loop. Therefore no commit-specific variables are defined yet. Functions or variables defined here can be used or modified in the following filter steps except the commit filter, for technical reasons.

--subdirectory-filter <directory>

Only look at the history which touches the given subdirectory. The result will contain that directory (and only that) as its project root. Implies Section 1, “Remap to ancestor”.

--env-filter <command>

This filter may be used if you only need to modify the environment in which the commit will be performed. Specifically, you might want to rewrite the author/committer name/email/time environment variables (see Section G.3.27, “git-commit-tree(1)” for details).

--tree-filter <command>

This is the filter for rewriting the tree and its contents. The argument is evaluated in shell with the working directory set to the root of the checked out tree. The new tree is then used as-is (new files are auto-added, disappeared files are auto-removed - neither .gitignore files nor any other ignore rules **HAVE ANY EFFECT**!).

--index-filter <command>

This is the filter for rewriting the index. It is similar to the tree filter but does not check out the tree, which makes it much faster. Frequently used with *git rm --cached --ignore-unmatch ...*, see EXAMPLES below. For hairy cases, see Section G.3.147, “git-update-index(1)”.

--parent-filter <command>

This is the filter for rewriting the commit's parent list. It will receive the parent string on stdin and shall output the new parent string on stdout. The parent string is in the format described in Section G.3.27, “git-commit-tree(1)”: empty for the initial commit, "-p parent" for a normal commit and "-p parent1 -p parent2 -p parent3 …" for a merge commit.

--msg-filter <command>

This is the filter for rewriting the commit messages. The argument is evaluated in the shell with the original commit message on standard input; its standard output is used as the new commit message.

--commit-filter <command>

This is the filter for performing the commit. If this filter is specified, it will be called instead of the *git commit-tree* command, with arguments of the form "<TREE_ID> [(-p <PARENT_COMMIT_ID>)…]" and the log message on stdin. The commit id is expected on stdout.

As a special extension, the commit filter may emit multiple commit ids; in that case, the rewritten children of the original commit will have all of them as parents.

You can use the *map* convenience function in this filter, and other convenience functions, too. For example, calling *skip_commit "$@"* will leave out the current commit (but not its changes! If you want that, use *git rebase* instead).

You can also use the *git_commit_non_empty_tree "$@"* instead of *git commit-tree "$@"* if you don't wish to keep commits with a single parent and that makes no change to the tree.

--tag-name-filter <command>

This is the filter for rewriting tag names. When passed, it will be called for every tag ref that points to a rewritten object (or to a tag object which points to a rewritten object). The original tag name is passed via standard input, and the new tag name is expected on standard output.

The original tags are not deleted, but can be overwritten; use "--tag-name-filter cat" to simply update the tags. In this case, be very careful and make sure you have the old tags backed up in case the conversion has run afoul.

Nearly proper rewriting of tag objects is supported. If the tag has a message attached, a new tag object will be created with the same message, author, and timestamp. If the tag has a signature attached, the signature will be stripped. It is by definition impossible to preserve signatures. The reason this is "nearly" proper, is because ideally if the tag did not change (points to the same object, has the same name, etc.) it should retain any signature. That is not the case, signatures will always be removed, buyer beware. There is also no support for changing the author or timestamp (or the tag message for that matter). Tags which point to other tags will be rewritten to point to the underlying commit.

--prune-empty

Some filters will generate empty commits that leave the tree untouched. This option instructs git-filter-branch to remove such commits if they have exactly one or zero non-pruned parents; merge commits will therefore

remain intact. This option cannot be used together with *--commit-filter*, though the same effect can be achieved by using the provided *git_commit_non_empty_tree* function in a commit filter.

--original <namespace>

Use this option to set the namespace where the original commits will be stored. The default value is *refs/original*.

-d <directory>

Use this option to set the path to the temporary directory used for rewriting. When applying a tree filter, the command needs to temporarily check out the tree to some directory, which may consume considerable space in case of large projects. By default it does this in the *.git-rewrite/* directory but you can override that choice by this parameter.

-f , --force

git filter-branch refuses to start with an existing temporary directory or when there are already refs starting with *refs/original/*, unless forced.

--state-branch <branch>

This option will cause the mapping from old to new objects to be loaded from named branch upon startup and saved as a new commit to that branch upon exit, enabling incremental of large trees. If *<branch>* does not exist it will be created.

<rev-list options>…

Arguments for *git rev-list*. All positive refs included by these options are rewritten. You may also specify options such as *--all*, but you must use -- to separate them from the *git filter-branch* options. Implies Section 1, “Remap to ancestor”.

1. Remap to ancestor

By using Section G.3.120, “git-rev-list(1)” arguments, e.g., path limiters, you can limit the set of revisions which get rewritten. However, positive refs on the command line are distinguished: we don't let them be excluded by such limiters. For this purpose, they are instead rewritten to point at the nearest ancestor that was not excluded.

EXIT STATUS

On success, the exit status is *0*. If the filter can't find any commits to rewrite, the exit status is *2*. On any other error, the exit status may be any other non-zero value.

EXAMPLES

Suppose you want to remove a file (containing confidential information or copyright violation) from all commits:

```
git filter-branch --tree-filter 'rm filename' HEAD
```

However, if the file is absent from the tree of some commit, a simple *rm filename* will fail for that tree and commit. Thus you may instead want to use *rm -f filename* as the script.

Using *--index-filter* with *git rm* yields a significantly faster version. Like with using *rm filename*, *git rm --cached filename* will fail if the file is absent from the tree of a commit. If you want to "completely forget" a file, it does not matter when it entered history, so we also add *--ignore-unmatch*:

```
git filter-branch --index-filter 'git rm --cached --ignore-unmatch
 filename' HEAD
```

Now, you will get the rewritten history saved in HEAD.

To rewrite the repository to look as if *foodir/* had been its project root, and discard all other history:

```
git filter-branch --subdirectory-filter foodir -- --all
```

Thus you can, e.g., turn a library subdirectory into a repository of its own. Note the -- that separates *filter-branch* options from revision options, and the *--all* to rewrite all branches and tags.

To set a commit (which typically is at the tip of another history) to be the parent of the current initial commit, in order to paste the other history behind the current history:

```
git filter-branch --parent-filter 'sed "s/^\$/-p <graft-id>/"' HEAD
```

(if the parent string is empty - which happens when we are dealing with the initial commit - add graftcommit as a parent). Note that this assumes history with a single root (that is, no merge without common ancestors happened). If this is not the case, use:

```
git filter-branch --parent-filter \
        'test $GIT_COMMIT = <commit-id> && echo "-p <graft-id>" || cat'
 HEAD
```

or even simpler:

```
git replace --graft $commit-id $graft-id
git filter-branch $graft-id..HEAD
```

To remove commits authored by "Darl McBribe" from the history:

```
git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_NAME" = "Darl McBribe" ];
        then
                skip_commit "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

The function *skip_commit* is defined as follows:

```
skip_commit()
{
        shift;
        while [ -n "$1" ];
        do
                shift;
                map "$1";
                shift;
        done;
}
```

The shift magic first throws away the tree id and then the -p parameters. Note that this handles merges properly! In case Darl committed a merge between P1 and P2, it will be propagated properly and all children of the merge will become merge commits with P1,P2 as their parents instead of the merge commit.

NOTE the changes introduced by the commits, and which are not reverted by subsequent commits, will still be in the rewritten branch. If you want to throw out *changes* together with the commits, you should use the interactive mode of *git rebase*.

You can rewrite the commit log messages using *--msg-filter*. For example, *git svn-id* strings in a repository created by *git svn* can be removed this way:

```
git filter-branch --msg-filter '
        sed -e "/^git-svn-id:/d"
'
```

If you need to add *Acked-by* lines to, say, the last 10 commits (none of which is a merge), use this command:

```
git filter-branch --msg-filter '
        cat &&
        echo "Acked-by: Bugs Bunny <bunny@bugzilla.org>"
' HEAD~10..HEAD
```

The *--env-filter* option can be used to modify committer and/or author identity. For example, if you found out that your commits have the wrong identity due to a misconfigured user.email, you can make a correction, before publishing the project, like this:

```
git filter-branch --env-filter '
        if test "$GIT_AUTHOR_EMAIL" = "root@localhost"
        then
                GIT_AUTHOR_EMAIL=john@example.com
        fi
        if test "$GIT_COMMITTER_EMAIL" = "root@localhost"
        then
                GIT_COMMITTER_EMAIL=john@example.com
        fi
' -- --all
```

To restrict rewriting to only part of the history, specify a revision range in addition to the new branch name. The new branch name will point to the top-most revision that a *git rev-list* of this range will print.

Consider this history:

```
     D--E--F--G--H
    /     /
A--B-----C
```

To rewrite only commits D,E,F,G,H, but leave A, B and C alone, use:

```
git filter-branch ... C..H
```

To rewrite commits E,F,G,H, use one of these:

```
git filter-branch ... C..H --not D
git filter-branch ... D..H --not C
```

To move the whole tree into a subdirectory, or remove it from there:

```
git filter-branch --index-filter \
        'git ls-files -s | sed "s-\t\"*-&newsubdir/-" |
                GIT_INDEX_FILE=$GIT_INDEX_FILE.new \
                        git update-index --index-info &&
         mv "$GIT_INDEX_FILE.new" "$GIT_INDEX_FILE"' HEAD
```

CHECKLIST FOR SHRINKING A REPOSITORY

git-filter-branch can be used to get rid of a subset of files, usually with some combination of *--index-filter* and *--subdirectory-filter*. People expect the resulting repository to be smaller than the original, but you need a few more steps to actually make it smaller, because Git tries hard not to lose your objects until you tell it to. First make sure that:

- You really removed all variants of a filename, if a blob was moved over its lifetime. *git log --name-only --follow --all -- filename* can help you find renames.
- You really filtered all refs: use *--tag-name-filter cat -- --all* when calling git-filter-branch.

Then there are two ways to get a smaller repository. A safer way is to clone, that keeps your original intact.

- Clone it with *git clone file:///path/to/repo*. The clone will not have the removed objects. See Section G.3.24, "git-clone(1)". (Note that cloning with a plain path just hardlinks everything!)

If you really don't want to clone it, for whatever reasons, check the following points instead (in this order). This is a very destructive approach, so **make a backup** or go back to cloning it. You have been warned.

- Remove the original refs backed up by git-filter-branch: say *git for-each-ref --format="%(refname)" refs/original/ | xargs -n 1 git update-ref -d*.
- Expire all reflogs with *git reflog expire --expire=now --all*.
- Garbage collect all unreferenced objects with *git gc --prune=now* (or if your git-gc is not new enough to support arguments to *--prune*, use *git repack -ad; git prune* instead).

PERFORMANCE

The performance of git-filter-branch is glacially slow; its design makes it impossible for a backward-compatible implementation to ever be fast:

- In editing files, git-filter-branch by design checks out each and every commit as it existed in the original repo. If your repo has *10^5* files and *10^5* commits, but each commit only modifies five files, then git-filter-branch will make you do *10^10* modifications, despite only having (at most) *5*10^5* unique blobs.
- If you try and cheat and try to make git-filter-branch only work on files modified in a commit, then two things happen
  - you run into problems with deletions whenever the user is simply trying to rename files (because attempting to delete files that don't exist looks like a no-op; it takes some chicanery to remap deletes across file renames when the renames happen via arbitrary user-provided shell)
  - even if you succeed at the map-deletes-for-renames chicanery, you still technically violate backward compatibility because users are allowed to filter files in ways that depend upon topology of commits instead of filtering solely based on file contents or names (though this has not been observed in the wild).
- Even if you don't need to edit files but only want to e.g. rename or remove some and thus can avoid checking out each file (i.e. you can use --index-filter), you still are passing shell snippets for your filters. This means that for every commit, you have to have a prepared git repo where those filters can be run. That's a significant setup.
- Further, several additional files are created or updated per commit by git-filter-branch. Some of these are for supporting the convenience functions provided by git-filter-branch (such as map()), while others are for keeping track of internal state (but could have also been accessed by user filters; one of git-filter-branch's regression tests does so). This essentially amounts to using the filesystem as an IPC mechanism between git-filter-branch and the user-provided filters. Disks tend to be a slow IPC mechanism, and writing these files also effectively represents a forced synchronization point between separate processes that we hit with every commit.
- The user-provided shell commands will likely involve a pipeline of commands, resulting in the creation of many processes per commit. Creating and running another process takes a widely varying amount of time between operating systems, but on any platform it is very slow relative to invoking a function.
- git-filter-branch itself is written in shell, which is kind of slow. This is the one performance issue that could be backward-compatibly fixed, but compared to the above problems that are intrinsic to the design of git-filter-branch, the language of the tool itself is a relatively minor issue.
  - Side note: Unfortunately, people tend to fixate on the written-in-shell aspect and periodically ask if git-filter-branch could be rewritten in another language to fix the performance issues. Not only does that ignore the bigger intrinsic problems with the design, it'd help less than you'd expect: if git-filter-branch itself were not shell, then the convenience functions (map(), skip_commit(), etc) and the *--setup* argument could no longer be executed once at the beginning of the program but would instead need to be prepended to every user filter (and thus re-executed with every commit).

The git filter-repo [https://github.com/newren/git-filter-repo/] tool is an alternative to git-filter-branch which does not suffer from these performance problems or the safety problems (mentioned below). For those with existing

tooling which relies upon git-filter-branch, *git filter-repo* also provides filter-lamely [https://github.com/newren/git-filter-repo/blob/master/contrib/filter-repo-demos/filter-lamely], a drop-in git-filter-branch replacement (with a few caveats). While filter-lamely suffers from all the same safety issues as git-filter-branch, it at least ameliorates the performance issues a little.

SAFETY

git-filter-branch is riddled with gotchas resulting in various ways to easily corrupt repos or end up with a mess worse than what you started with:

- Someone can have a set of "working and tested filters" which they document or provide to a coworker, who then runs them on a different OS where the same commands are not working/tested (some examples in the git-filter-branch manpage are also affected by this). BSD vs. GNU userland differences can really bite. If lucky, error messages are spewed. But just as likely, the commands either don't do the filtering requested, or silently corrupt by making some unwanted change. The unwanted change may only affect a few commits, so it's not necessarily obvious either. (The fact that problems won't necessarily be obvious means they are likely to go unnoticed until the rewritten history is in use for quite a while, at which point it's really hard to justify another flag-day for another rewrite.)

- Filenames with spaces are often mishandled by shell snippets since they cause problems for shell pipelines. Not everyone is familiar with find -print0, xargs -0, git-ls-files -z, etc. Even people who are familiar with these may assume such flags are not relevant because someone else renamed any such files in their repo back before the person doing the filtering joined the project. And often, even those familiar with handling arguments with spaces may not do so just because they aren't in the mindset of thinking about everything that could possibly go wrong.

- Non-ascii filenames can be silently removed despite being in a desired directory. Keeping only wanted paths is often done using pipelines like *git ls-files | grep -v ^WANTED_DIR/ | xargs git rm*. ls-files will only quote filenames if needed, so folks may not notice that one of the files didn't match the regex (at least not until it's much too late). Yes, someone who knows about core.quotePath can avoid this (unless they have other special characters like \t, \n, or "), and people who use ls-files -z with something other than grep can avoid this, but that doesn't mean they will.

- Similarly, when moving files around, one can find that filenames with non-ascii or special characters end up in a different directory, one that includes a double quote character. (This is technically the same issue as above with quoting, but perhaps an interesting different way that it can and has manifested as a problem.)

- It's far too easy to accidentally mix up old and new history. It's still possible with any tool, but git-filter-branch almost invites it. If lucky, the only downside is users getting frustrated that they don't know how to shrink their repo and remove the old stuff. If unlucky, they merge old and new history and end up with multiple "copies" of each commit, some of which have unwanted or sensitive files and others which don't. This comes about in multiple different ways:

  - the default to only doing a partial history rewrite (*--all* is not the default and few examples show it)

  - the fact that there's no automatic post-run cleanup

  - the fact that --tag-name-filter (when used to rename tags) doesn't remove the old tags but just adds new ones with the new name

  - the fact that little educational information is provided to inform users of the ramifications of a rewrite and how to avoid mixing old and new history. For example, this man page discusses how users need to understand that they need to rebase their changes for all their branches on top of new history (or delete and reclone), but that's only one of multiple concerns to consider. See the "DISCUSSION" section of the git filter-repo manual page for more details.

- Annotated tags can be accidentally converted to lightweight tags, due to either of two issues:

  - Someone can do a history rewrite, realize they messed up, restore from the backups in refs/original/, and then redo their git-filter-branch command. (The backup in refs/original/ is not a real backup; it dereferences tags first.)

  - Running git-filter-branch with either --tags or --all in your <rev-list-options>. In order to retain annotated tags as annotated, you must use --tag-name-filter (and must not have restored from refs/original/ in a previously botched rewrite).

- Any commit messages that specify an encoding will become corrupted by the rewrite; git-filter-branch ignores the encoding, takes the original bytes, and feeds it to commit-tree without telling it the proper encoding. (This happens whether or not --msg-filter is used.)

- Commit messages (even if they are all UTF-8) by default become corrupted due to not being updated -- any references to other commit hashes in commit messages will now refer to no-longer-extant commits.

- There are no facilities for helping users find what unwanted crud they should delete, which means they are much more likely to have incomplete or partial cleanups that sometimes result in confusion and people wasting time trying to understand. (For example, folks tend to just look for big files to delete instead of big directories or extensions, and once they do so, then sometime later folks using the new repository who are going through history will notice a build artifact directory that has some files but not others, or a cache of dependencies (node_modules or similar) which couldn't have ever been functional since it's missing some files.)

- If --prune-empty isn't specified, then the filtering process can create hoards of confusing empty commits

- If --prune-empty is specified, then intentionally placed empty commits from before the filtering operation are also pruned instead of just pruning commits that became empty due to filtering rules.

- If --prune-empty is specified, sometimes empty commits are missed and left around anyway (a somewhat rare bug, but it happens…)

- A minor issue, but users who have a goal to update all names and emails in a repository may be led to --env-filter which will only update authors and committers, missing taggers.

- If the user provides a --tag-name-filter that maps multiple tags to the same name, no warning or error is provided; git-filter-branch simply overwrites each tag in some undocumented pre-defined order resulting in only one tag at the end. (A git-filter-branch regression test requires this surprising behavior.)

Also, the poor performance of git-filter-branch often leads to safety issues:

- Coming up with the correct shell snippet to do the filtering you want is sometimes difficult unless you're just doing a trivial modification such as deleting a couple files. Unfortunately, people often learn if the snippet is right or wrong by trying it out, but the rightness or wrongness can vary depending on special circumstances (spaces in filenames, non-ascii filenames, funny author names or emails, invalid timezones, presence of grafts or replace objects, etc.), meaning they may have to wait a long time, hit an error, then restart. The performance of git-filter-branch is so bad that this cycle is painful, reducing the time available to carefully re-check (to say nothing about what it does to the patience of the person doing the rewrite even if they do technically have more time available). This problem is extra compounded because errors from broken filters may not be shown for a long time and/or get lost in a sea of output. Even worse, broken filters often just result in silent incorrect rewrites.

- To top it all off, even when users finally find working commands, they naturally want to share them. But they may be unaware that their repo didn't have some special cases that someone else's does. So, when someone else with a different repository runs the same commands, they get hit by the problems above. Or, the user just runs commands that really were vetted for special cases, but they run it on a different OS where it doesn't work, as noted above.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.51. git-fmt-merge-msg(1)

2024-07-02

Revision History

2024-07-02

NAME

git-fmt-merge-msg - Produce a merge commit message

SYNOPSIS

git fmt-merge-msg [-m <message>] [--into-name <branch>] [--log[=<n>] | --no-log]
git fmt-merge-msg [-m <message>] [--log[=<n>] | --no-log] -F <file>

DESCRIPTION

Takes the list of merged objects on stdin and produces a suitable commit message to be used for the merge commit, usually to be passed as the <*merge-message*> argument of *git merge*.

This command is intended mostly for internal use by scripts automatically invoking *git merge*.

OPTIONS

--log[=<n>]

In addition to branch names, populate the log message with one-line descriptions from the actual commits that are being merged. At most <n> commits from each merge parent will be used (20 if <n> is omitted). This overrides the *merge.log* configuration variable.

--no-log

Do not list one-line descriptions from the actual commits being merged.

--[no-]summary

Synonyms to --log and --no-log; these are deprecated and will be removed in the future.

-m <message> , --message <message>

Use <message> instead of the branch names for the first line of the log message. For use with --*log*.

--into-name <branch>

Prepare the merge message as if merging to the branch <*branch*>, instead of the name of the real branch to which the merge is made.

-F <file> , --file <file>

Take the list of merged objects from <file> instead of stdin.

CONFIGURATION

merge.branchdesc

In addition to branch names, populate the log message with the branch description text associated with them. Defaults to false.

merge.log

In addition to branch names, populate the log message with at most the specified number of one-line descriptions from the actual commits that are being merged. Defaults to false, and true is a synonym for 20.

merge.suppressDest

By adding a glob that matches the names of integration branches to this multi-valued configuration variable, the default merge message computed for merges into these integration branches will omit "into <branch name>" from its title.

An element with an empty value can be used to clear the list of globs accumulated from previous configuration entries. When there is no *merge.suppressDest* variable defined, the default value of *master* is used for backward compatibility.

merge.summary

Synonym to *merge.log*; this is deprecated and will be removed in the future.

EXAMPLES

```
$ git fetch origin master
$ git fmt-merge-msg --log <$GIT_DIR/FETCH_HEAD
```

Print a log message describing a merge of the "master" branch from the "origin" remote.

SEE ALSO

Section G.3.86, “git-merge(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.52. git-for-each-ref(1)

2024-07-02

Revision History

2024-07-02

NAME

git-for-each-ref - Output information on each ref

SYNOPSIS

git for-each-ref [--count=<count>] [--shell|--perl|--python|--tcl]
[(--sort=<key>)…] [--format=<format>]
[--include-root-refs] [--stdin | <pattern>…]
[--points-at=<object>]
[--merged[=<object>]] [--no-merged[=<object>]]
[--contains[=<object>]] [--no-contains[=<object>]]
[--exclude=<pattern> …]

DESCRIPTION

Iterate over all refs that match <*pattern*> and show them according to the given <*format*>, after sorting them according to the given set of <*key*>. If <*count*> is given, stop after showing that many refs. The interpolated values in <*format*> can optionally be quoted as string literals in the specified host language allowing their direct evaluation in that language.

OPTIONS

<pattern>…

If one or more patterns are given, only refs are shown that match against at least one pattern, either using fnmatch(3) or literally, in the latter case matching completely or from the beginning up to a slash.

--stdin

If *--stdin* is supplied, then the list of patterns is read from standard input instead of from the argument list.

--count=<count>

By default the command shows all refs that match *<pattern>*. This option makes it stop after showing that many refs.

--sort=<key>

A field name to sort on. Prefix - to sort in descending order of the value. When unspecified, *refname* is used. You may use the --sort=<key> option multiple times, in which case the last key becomes the primary key.

--format=<format>

A string that interpolates *%(fieldname)* from a ref being shown and the object it points at. In addition, the string literal *%%* renders as *%* and *%xx* - where *xx* are hex digits - renders as the character with hex code *xx*. For example, *%00* interpolates to *\0* (NUL), *%09* to *\t* (TAB), and *%0a* to *\n* (LF).

When unspecified, *<format>* defaults to *%(objectname) SPC %(objecttype) TAB %(refname)*.

--color[=<when>]

Respect any colors specified in the *--format* option. The *<when>* field must be one of *always*, *never*, or *auto* (if *<when>* is absent, behave as if *always* was given).

--shell , --perl , --python , --tcl

If given, strings that substitute *%(fieldname)* placeholders are quoted as string literals suitable for the specified host language. This is meant to produce a scriptlet that can directly be `eval`ed.

--points-at=<object>

Only list refs which points at the given object.

--merged[=<object>]

Only list refs whose tips are reachable from the specified commit (HEAD if not specified).

--no-merged[=<object>]

Only list refs whose tips are not reachable from the specified commit (HEAD if not specified).

--contains[=<object>]

Only list refs which contain the specified commit (HEAD if not specified).

--no-contains[=<object>]

Only list refs which don't contain the specified commit (HEAD if not specified).

--ignore-case

Sorting and filtering refs are case insensitive.

--omit-empty

Do not print a newline after formatted refs where the format expands to the empty string.

--exclude=<pattern>

If one or more patterns are given, only refs which do not match any excluded pattern(s) are shown. Matching is done using the same rules as *<pattern>* above.

--include-root-refs

List root refs (HEAD and pseudorefs) apart from regular refs.

FIELD NAMES

Various values from structured fields in referenced objects can be used to interpolate into the resulting output, or as sort keys.

For all objects, the following names can be used:

refname

The name of the ref (the part after $GIT_DIR/). For a non-ambiguous short name of the ref append *:short*. The option core.warnAmbiguousRefs is used to select the strict abbreviation mode. If *lstrip=<N>* (*rstrip=<N>*) is appended, strips *<N>* slash-separated path components from the front (back) of the refname (e.g. *%(refname:lstrip=2)* turns *refs/tags/foo* into *foo* and *%(refname:rstrip=2)* turns *refs/tags/foo* into *refs*). If *<N>* is a negative number, strip as many path components as necessary from the specified end to leave *-<N>* path components (e.g. *%(refname:lstrip=-2)* turns *refs/tags/foo* into *tags/foo* and *%(refname:rstrip=-1)* turns *refs/tags/foo* into *refs*). When the ref does not have enough components, the result becomes an empty string if stripping with positive <N>, or it becomes the full refname if stripping with negative <N>. Neither is an error.

strip can be used as a synonym to *lstrip*.

objecttype

The type of the object (*blob*, *tree*, *commit*, *tag*).

objectsize

The size of the object (the same as *git cat-file -s* reports). Append *:disk* to get the size, in bytes, that the object takes up on disk. See the note about on-disk sizes in the *CAVEATS* section below.

objectname

The object name (aka SHA-1). For a non-ambiguous abbreviation of the object name append *:short*. For an abbreviation of the object name with desired length append *:short=<length>*, where the minimum length is MINIMUM_ABBREV. The length may be exceeded to ensure unique object names.

deltabase

This expands to the object name of the delta base for the given object, if it is stored as a delta. Otherwise it expands to the null object name (all zeroes).

upstream

The name of a local ref which can be considered upstream from the displayed ref. Respects *:short*, *:lstrip* and *:rstrip* in the same way as *refname* above. Additionally respects *:track* to show "[ahead N, behind M]" and *:trackshort* to show the terse version: ">" (ahead), "<" (behind), "<>" (ahead and behind), or "=" (in sync). *:track* also prints "[gone]" whenever unknown upstream ref is encountered. Append *:track,nobracket* to show tracking information without brackets (i.e "ahead N, behind M").

For any remote-tracking branch *%(upstream)*, *%(upstream:remotename)* and *%(upstream:remoteref)* refer to the name of the remote and the name of the tracked remote ref, respectively. In other words, the remote-tracking branch can be updated explicitly and individually by using the refspec *%(upstream:remoteref):%(upstream)* to fetch from *%(upstream:remotename)*.

Has no effect if the ref does not have tracking information associated with it. All the options apart from *nobracket* are mutually exclusive, but if used together the last option is selected.

push

The name of a local ref which represents the *@{push}* location for the displayed ref. Respects *:short*, *:lstrip*, *:rstrip*, *:track*, *:trackshort*, *:remotename*, and *:remoteref* options as *upstream* does. Produces an empty string if no *@{push}* ref is configured.

HEAD

* if HEAD matches current ref (the checked out branch), ' ' otherwise.

color

Change output color. Followed by *:<colorname>*, where color names are described under Values in the "CONFIGURATION FILE" section of Section G.3.29, “git-config(1)”. For example, *%(color:bold red)*.

align

Left-, middle-, or right-align the content between %(align:…) and %(end). The "align:" is followed by *width=<width>* and *position=<position>* in any order separated by a comma, where the *<position>* is either left, right or middle, default being left and *<width>* is the total length of the content with alignment. For brevity, the "width=" and/or "position=" prefixes may be omitted, and bare <width> and <position> used instead. For instance, *%(align:<width>,<position>)*. If the contents length is more than the width then no alignment is performed. If used with *--quote* everything in between %(align:…) and %(end) is quoted, but if nested then only the topmost level performs quoting.

if

Used as %(if)…%(then)…%(end) or %(if)…%(then)…%(else)…%(end). If there is an atom with value or string literal after the %(if) then everything after the %(then) is printed, else if the %(else) atom is used, then everything after %(else) is printed. We ignore space when evaluating the string before %(then), this is useful when we use the %(HEAD) atom which prints either "*" or " " and we want to apply the *if* condition only on the *HEAD* ref. Append ":equals=<string>" or ":notequals=<string>" to compare the value between the %(if:…) and %(then) atoms with the given string.

symref

The ref which the given symbolic ref refers to. If not a symbolic ref, nothing is printed. Respects the *:short*, *:lstrip* and *:rstrip* options in the same way as *refname* above.

signature

The GPG signature of a commit.

signature:grade

Show "G" for a good (valid) signature, "B" for a bad signature, "U" for a good signature with unknown validity, "X" for a good signature that has expired, "Y" for a good signature made by an expired key, "R" for a good signature made by a revoked key, "E" if the signature cannot be checked (e.g. missing key) and "N" for no signature.

signature:signer

The signer of the GPG signature of a commit.

signature:key

The key of the GPG signature of a commit.

signature:fingerprint

The fingerprint of the GPG signature of a commit.

signature:primarykeyfingerprint

The primary key fingerprint of the GPG signature of a commit.

signature:trustlevel

The trust level of the GPG signature of a commit. Possible outputs are *ultimate*, *fully*, *marginal*, *never* and *undefined*.

worktreepath

The absolute path to the worktree in which the ref is checked out, if it is checked out in any linked worktree. Empty string otherwise.

ahead-behind:<committish>

Two integers, separated by a space, demonstrating the number of commits ahead and behind, respectively, when comparing the output ref to the *<committish>* specified in the format.

describe[:options]

A human-readable name, like Section G.3.39, “git-describe(1)”; empty string for undescribable commits. The *describe* string may be followed by a colon and one or more comma-separated options.

tags=<bool-value>

Instead of only considering annotated tags, consider lightweight tags as well; see the corresponding option in Section G.3.39, “git-describe(1)” for details.

abbrev=<number>

Use at least <number> hexadecimal digits; see the corresponding option in Section G.3.39, “git-describe(1)” for details.

match=<pattern>

Only consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix; see the corresponding option in Section G.3.39, “git-describe(1)” for details.

exclude=<pattern>

Do not consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix; see the corresponding option in Section G.3.39, “git-describe(1)” for details.

In addition to the above, for commit and tag objects, the header field names (*tree*, *parent*, *object*, *type*, and *tag*) can be used to specify the value in the header field. Fields *tree* and *parent* can also be used with modifier *:short* and *:short=<length>* just like *objectname*.

For commit and tag objects, the special *creatordate* and *creator* fields will correspond to the appropriate date or name-email-date tuple from the *committer* or *tagger* fields depending on the object type. These are intended for working on a mix of annotated and lightweight tags.

For tag objects, a *fieldname* prefixed with an asterisk (*) expands to the *fieldname* value of the peeled object, rather than that of the tag object itself.

Fields that have name-email-date tuple as its value (*author*, *committer*, and *tagger*) can be suffixed with *name*, *email*, and *date* to extract the named component. For email fields (*authoremail*, *committeremail* and *taggeremail*), *:trim* can be appended to get the email without angle brackets, and *:localpart* to get the part before the @ symbol out of the trimmed email. In addition to these, the *:mailmap* option and the corresponding *:mailmap,trim* and *:mailmap,localpart* can be used (order does not matter) to get values of the name and email according to the .mailmap file or according to the file set in the mailmap.file or mailmap.blob configuration variable (see Section G.4.9, “gitmailmap(5)”).

The raw data in an object is *raw*.

raw:size

The raw data size of the object.

Note that *--format=%(raw)* can not be used with *--python*, *--shell*, *--tcl*, because such language may not support arbitrary binary data in their string variable type.

The message in a commit or a tag object is *contents*, from which *contents:<part>* can be used to extract various parts out of:

contents:size

The size in bytes of the commit or tag message.

contents:subject

The first paragraph of the message, which typically is a single line, is taken as the "subject" of the commit or the tag message. Instead of *contents:subject*, field *subject* can also be used to obtain same results. *:sanitize* can be appended to *subject* for subject line suitable for filename.

contents:body

The remainder of the commit or the tag message that follows the "subject".

contents:signature

The optional GPG signature of the tag.

contents:lines=N

The first *N* lines of the message.

Additionally, the trailers as interpreted by Section G.3.73, “git-interpret-trailers(1)” are obtained as *trailers[:options]* (or by using the historical alias *contents:trailers[:options]*). For valid [:option] values see *trailers* section of Section G.3.74, “git-log(1)”.

For sorting purposes, fields with numeric values sort in numeric order (*objectsize*, *authordate*, *committerdate*, *creatordate*, *taggerdate*). All other fields are used to sort in their byte-value order.

There is also an option to sort by versions, this can be done by using the fieldname *version:refname* or its alias *v:refname*.

In any case, a field name that refers to a field inapplicable to the object referred by the ref does not cause an error. It returns an empty string instead.

As a special case for the date-type fields, you may specify a format for the date by adding *:* followed by date format name (see the values the *--date* option to Section G.3.120, “git-rev-list(1)” takes). If this formatting is provided in a *--sort* key, references will be sorted according to the byte-value of the formatted string rather than the numeric value of the underlying timestamp.

Some atoms like %(align) and %(if) always require a matching %(end). We call them "opening atoms" and sometimes denote them as %($open).

When a scripting language specific quoting is in effect, everything between a top-level opening atom and its matching %(end) is evaluated according to the semantics of the opening atom and only its result from the top-level is quoted.

EXAMPLES

An example directly producing formatted text. Show the most recent 3 tagged commits:

```
#!/bin/sh

git for-each-ref --count=3 --sort='-*authordate' \
--format='From: %(*authorname) %(*authoremail)
Subject: %(*subject)
Date: %(*authordate)
Ref: %(*refname)
```

```
%(*body)
' 'refs/tags'
```

A simple example showing the use of shell eval on the output, demonstrating the use of --shell. List the prefixes of all heads:

```
#!/bin/sh

git for-each-ref --shell --format="ref=%(refname)" refs/heads | \
while read entry
do
        eval "$entry"
        echo `dirname $ref`
done
```

A bit more elaborate report on tags, demonstrating that the format may be an entire script:

```
#!/bin/sh

fmt='
        r=%(refname)
        t=%(*objecttype)
        T=${r#refs/tags/}

        o=%(*objectname)
        n=%(*authorname)
        e=%(*authoremail)
        s=%(*subject)
        d=%(*authordate)
        b=%(*body)

        kind=Tag
        if test "z$t" = z
        then
                # could be a lightweight tag
                t=%(objecttype)
                kind="Lightweight tag"
                o=%(objectname)
                n=%(authorname)
                e=%(authoremail)
                s=%(subject)
                d=%(authordate)
                b=%(body)
        fi
        echo "$kind $T points at a $t object $o"
        if test "z$t" = zcommit
        then
                echo "The commit was authored by $n $e
at $d, and titled

    $s

Its message reads as:
"
                echo "$b" | sed -e "s/^/    /"
                echo
        fi
'
```

```
eval=`git for-each-ref --shell --format="$fmt" \
        --sort='*objecttype' \
        --sort=-taggerdate \
        refs/tags`
eval "$eval"
```

An example to show the usage of %(if)…%(then)…%(else)…%(end). This prefixes the current branch with a star.

```
git for-each-ref --format="%(if)%(HEAD)%(then)* %(else)
 %(end)%(refname:short)" refs/heads/
```

An example to show the usage of %(if)…%(then)…%(end). This prints the authorname, if present.

```
git for-each-ref --format="%(refname)%(if)%(authorname)%(then) Authored by:
 %(authorname)%(end)"
```

CAVEATS

Note that the sizes of objects on disk are reported accurately, but care should be taken in drawing conclusions about which refs or objects are responsible for disk usage. The size of a packed non-delta object may be much larger than the size of objects which delta against it, but the choice of which object is the base and which is the delta is arbitrary and is subject to change during a repack.

Note also that multiple copies of an object may be present in the object database; in this case, it is undefined which copy's size or delta base will be reported.

NOTES

When combining multiple *--contains* and *--no-contains* filters, only references that contain at least one of the *--contains* commits and contain none of the *--no-contains* commits are shown.

When combining multiple *--merged* and *--no-merged* filters, only references that are reachable from at least one of the *--merged* commits and from none of the *--no-merged* commits are shown.

SEE ALSO

Section G.3.133, “git-show-ref(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.53. git-for-each-repo(1)

2024-07-02

Revision History

2024-07-02

NAME

git-for-each-repo - Run a Git command on a list of repositories

SYNOPSIS

git for-each-repo --config=<config> [--] <arguments>

DESCRIPTION

Run a Git command on a list of repositories. The arguments after the known options or -- indicator are used as the arguments for the Git subprocess.

THIS COMMAND IS EXPERIMENTAL. THE BEHAVIOR MAY CHANGE.

For example, we could run maintenance on each of a list of repositories stored in a *maintenance.repo* config variable using

```
git for-each-repo --config=maintenance.repo maintenance run
```

This will run *git -C <repo> maintenance run* for each value *<repo>* in the multi-valued config variable *maintenance.repo*.

OPTIONS

--config=<config>

Use the given config variable as a multi-valued list storing absolute path names. Iterate on that list of paths to run the given arguments.

These config values are loaded from system, global, and local Git config, as available. If *git for-each-repo* is run in a directory that is not a Git repository, then only the system and global config is used.

--keep-going

Continue with the remaining repositories if the command failed on a repository. The exit code will still indicate that the overall operation was not successful.

Note that the exact exit code of the failing command is not passed through as the exit code of the *for-each-repo* command: If the command failed in any of the specified repositories, the overall exit code will be 1.

SUBPROCESS BEHAVIOR

If any *git -C <repo> <arguments>* subprocess returns a non-zero exit code, then the *git for-each-repo* process returns that exit code without running more subprocesses.

Each *git -C <repo> <arguments>* subprocess inherits the standard file descriptors *stdin*, *stdout*, and *stderr*.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.54. git-format-patch(1)

2024-07-02
Revision History
2024-07-02

NAME

git-format-patch - Prepare patches for e-mail submission

SYNOPSIS

git format-patch [-k] [(-o|--output-directory) <dir> | --stdout]
[--no-thread | --thread[=<style>]]
[(--attach|--inline)[=<boundary>] | --no-attach]
[-s | --signoff]
[--signature=<signature> | --no-signature]
[--signature-file=<file>]
[-n | --numbered | -N | --no-numbered]
[--start-number <n>] [--numbered-files]
[--in-reply-to=<message-id>] [--suffix=.<sfx>]
[--ignore-if-in-upstream] [--always]

[--cover-from-description=<mode>]
[--rfc] [--subject-prefix=<subject-prefix>]
[(--reroll-count|-v) <n>]
[--to=<email>] [--cc=<email>]
[--[no-]cover-letter] [--quiet]
[--[no-]encode-email-headers]
[--no-notes | --notes[=<ref>]]
[--interdiff=<previous>]
[--range-diff=<previous> [--creation-factor=<percent>]]
[--filename-max-length=<n>]
[--progress]
[<common-diff-options>]
[<since> | <revision-range>]

DESCRIPTION

Prepare each non-merge commit with its "patch" in one "message" per commit, formatted to resemble a UNIX mailbox. The output of this command is convenient for e-mail submission or for use with *git am*.

A "message" generated by the command consists of three parts:

- A brief metadata header that begins with *From <commit>* with a fixed *Mon Sep 17 00:00:00 2001* datestamp to help programs like "file(1)" to recognize that the file is an output from this command, fields that record the author identity, the author date, and the title of the change (taken from the first paragraph of the commit log message).
- The second and subsequent paragraphs of the commit log message.
- The "patch", which is the "diff -p --stat" output (see Section G.3.44, “git-diff(1)”) between the commit and its parent.

The log message and the patch are separated by a line with a three-dash line.

There are two ways to specify which commits to operate on.

1. A single commit, <since>, specifies that the commits leading to the tip of the current branch that are not in the history that leads to the <since> to be output.
2. Generic <revision-range> expression (see "SPECIFYING REVISIONS" section in Section G.4.14, “gitrevisions(7)”) means the commits in the specified range.

The first rule takes precedence in the case of a single <commit>. To apply the second rule, i.e., format everything since the beginning of history up until <commit>, use the *--root* option: *git format-patch --root <commit>*. If you want to format only <commit> itself, you can do this with *git format-patch -1 <commit>*.

By default, each output file is numbered sequentially from 1, and uses the first line of the commit message (massaged for pathname safety) as the filename. With the *--numbered-files* option, the output file names will only be numbers, without the first line of the commit appended. The names of the output files are printed to standard output, unless the *--stdout* option is specified.

If *-o* is specified, output files are created in <dir>. Otherwise they are created in the current working directory. The default path can be set with the *format.outputDirectory* configuration option. The *-o* option takes precedence over *format.outputDirectory*. To store patches in the current working directory even when *format.outputDirectory* points elsewhere, use *-o .*. All directory components will be created.

By default, the subject of a single patch is "[PATCH] " followed by the concatenation of lines from the commit message up to the first blank line (see the DISCUSSION section of Section G.3.28, “git-commit(1)”).

When multiple patches are output, the subject prefix will instead be "[PATCH n/m] ". To force 1/1 to be added for a single patch, use *-n*. To omit patch numbers from the subject, use *-N*.

If given *--thread*, *git-format-patch* will generate *In-Reply-To* and *References* headers to make the second and subsequent patch mails appear as replies to the first mail; this also generates a *Message-ID* header to reference.

OPTIONS

-p , --no-stat

Generate plain patches without any diffstats.

-U<n> , --unified=<n>

Generate diffs with <n> lines of context instead of the usual three.

--output=<file>

Output to a specific file instead of stdout.

--output-indicator-new=<char> , --output-indicator-old=<char> , --output-indicator-context=<char>

Specify the character used to indicate new, old or context lines in the generated patch. Normally they are +, - and ' ' respectively.

--indent-heuristic

Enable the heuristic that shifts diff hunk boundaries to make patches easier to read. This is the default.

--no-indent-heuristic

Disable the indent heuristic.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--anchored=<text>

Generate a diff using the "anchored diff" algorithm.

This option may be specified more than once.

If a line exists in both the source and destination, exists only once, and starts with this text, this algorithm attempts to prevent it from appearing as a deletion or addition in the output. It uses the "patience diff" algorithm internally.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

default, myers

The basic greedy diff algorithm. Currently, this is the default.

minimal

Spend extra time to make sure the smallest possible diff is produced.

patience

Use "patience diff" algorithm when generating patches.

histogram

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured the *diff.algorithm* variable to a non-default value and want to use the default one, then you have to use *--diff-algorithm=default* option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by *<width>*. The width of the filename part can be limited by giving another width *<name-width>* after a comma or by setting *diff.statNameWidth=<width>*. The width of the graph part can be limited by using *--stat-graph-width=<width>* or by setting *diff.statGraphWidth=<width>*. Using *--stat* or *--stat-graph-width* affects all commands generating a stat graph, while setting *diff.statNameWidth* or *diff.statGraphWidth* does not affect *git format-patch*. By giving a third parameter *<count>*, you can limit the output to the first *<count>* lines, followed by ... if there are more.

These parameters can also be set individually with *--stat-width=<width>*, *--stat-name-width=<name-width>* and *--stat-count=<count>*.

--compact-summary

Output a condensed summary of extended header information such as file creations or deletions ("new" or "gone", optionally "+l" if it's a symlink) and mode changes ("+x" or "-x" for adding or removing executable bit respectively) in diffstat. The information is put between the filename part and the graph part. Implies *--stat*.

--numstat

Similar to *--stat*, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two - instead of saying *0 0*.

--shortstat

Output only the last line of the *--stat* format containing total number of modified files, as well as number of added and deleted lines.

-X[<param1,param2,…>] , --dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of *--dirstat* can be customized by passing it a comma separated list of parameters. The defaults are controlled by the *diff.dirstat* configuration variable (see Section G.3.29, “git-config(1)”). The following parameters are available:

changes

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

lines

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive *--dirstat* behavior than the *changes* behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other *--*stat* options.

files

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest *--dirstat* behavior, since it does not have to look at the file contents at all.

cumulative

Count changes in a child directory for the parent directory as well. Note that when using *cumulative*, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the *noncumulative* parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: *--dirstat=files,10,cumulative*.

--cumulative

Synonym for --dirstat=cumulative

--dirstat-by-file[=<param1,param2>…]

Synonym for --dirstat=files,<param1>,<param2>…

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--[no-]rename-empty

Whether to use empty blobs as rename source.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to *--full-index*, output a binary diff that can be applied with *git-apply*.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show the shortest prefix that is at least *<n>* hexdigits long that uniquely refers the object. In diff-patch output format, *--full-index* takes higher precedence, i.e. if *--full-index* is specified, full blob names will be shown regardless of *--abbrev*. Non default number of digits can be specified with *--abbrev=<n>*.

-B[<n>][/<m>] , --break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion

of everything old followed by a single insertion of everything new, and the number *m* controls this aspect of the -B option (defaults to 60%). *-B/70%* specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number *n* controls this aspect of the -B option (defaults to 50%). *-B20%* specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>] , --find-renames[=<n>]

Detect renames. If *n* is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, *-M90%* means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a *%* sign, the number is to be read as a fraction, with a decimal point before it. I.e., *-M5* becomes 0.5, and is thus the same as *-M50%*. Similarly, *-M05* is the same as *-M5%*. To limit detection to exact renames, use *-M100%*. The default similarity index is 50%.

-C[<n>] , --find-copies[=<n>]

Detect copies as well as renames. See also *--find-copies-harder*. If *n* is specified, it has the same meaning as for *-M<n>*.

--find-copies-harder

For performance reasons, by default, *-C* option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one *-C* option has the same effect.

-D , --irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and */dev/null*. The resulting patch is not meant to be applied with *patch* or *git apply*; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lacks enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with *-B*, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The *-M* and *-C* options involve some preliminary steps that can detect subsets of renames/copies cheaply, followed by an exhaustive fallback portion that compares all remaining unpaired destinations to all relevant sources. (For renames, only remaining unpaired sources are relevant; for copies, all original sources are relevant.) For N sources and destinations, this exhaustive check is O(N^2). This option prevents the exhaustive portion of rename/copy detection from running if the number of source/destination files involved exceeds the specified number. Defaults to diff.renameLimit. Note that a value of 0 is treated as unlimited.

-O<orderfile>

Control the order in which files appear in the output. This overrides the *diff.orderFile* configuration variable (see Section G.3.29, “git-config(1)”). To cancel *diff.orderFile*, use *-O/dev/null*.

The output order is determined by the order of glob patterns in <orderfile>. All files with pathnames that match the first pattern are output first, all files with pathnames that match the second pattern (but not the first) are output next, and so on. All files with pathnames that do not match any pattern are output last, as if there was an implicit match-all pattern at the end of the file. If multiple pathnames have the same rank (they match the same pattern but no earlier patterns), their output order relative to each other is the normal order.

<orderfile> is parsed as follows:

- Blank lines are ignored, so they can be used as separators for readability.

- Lines starting with a hash ("#") are ignored, so they can be used for comments. Add a backslash ("\") to the beginning of the pattern if it starts with a hash.
- Each other line contains a single pattern.

Patterns have the same syntax and semantics as patterns used for fnmatch(3) without the FNM_PATHNAME flag, except a pathname also matches a pattern if removing any number of the final pathname components matches the pattern. For example, the pattern "*foo*bar*" matches "*fooasdfbar*" and "*foo/bar/baz/asdf*" but not "*foobarx*".

--skip-to=<file> , --rotate-to=<file>

Discard the files before the named <file> from the output (i.e. *skip to*), or move them to the end of the output (i.e. *rotate to*). These options were invented primarily for the use of the *git difftool* command, and may not be very useful otherwise.

--relative[=<path>] , --no-relative

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument. *--no-relative* can be used to countermand both *diff.relative* config option and previous *--relative*.

-a , --text

Treat all files as text.

--ignore-cr-at-eol

Ignore carriage-return at the end of line when doing a comparison.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b , --ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w , --ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

-I<regex> , --ignore-matching-lines=<regex>

Ignore changes whose all lines match <regex>. This option may be specified more than once.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other. Defaults to *diff.interHunkContext* or 0 if the config option is unset.

-W , --function-context

Show whole function as context lines for each change. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with Section G.4.2, “gitattributes(5)”, you need to use this option with Section G.3.74, “git-log(1)” and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv , --no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See Section G.4.2, “gitattributes(5)” for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for Section G.3.44, “git-diff(1)” and Section G.3.74, “git-log(1)”, but not for Section G.3.54, “git-format-patch(1)” or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in Section G.3.29, “git-config(1)” or Section G.4.10, “gitmodules(5)”. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

--default-prefix

Use the default source and destination prefixes ("a/" and "b/"). This overrides configuration variables such as *diff.noprefix*, *diff.srcPrefix*, *diff.dstPrefix*, and *diff.mnemonicPrefix* (see *git-config*(1)).

--line-prefix=<prefix>

Prepend an additional prefix to every line of output.

--ita-invisible-in-index

By default entries added by "git add -N" appear as an existing empty file in "git diff" and a new file in "git diff --cached". This option makes the entry appear as a new file in "git diff" and non-existent in "git diff --cached". This option could be reverted with *--ita-visible-in-index*. Both options are experimental and could be removed in future.

For more detailed explanation on these common options, see also Section G.4.4, “gitdiffcore(7)”.

-<n>

Prepare patches from the topmost <n> commits.

-o <dir> , --output-directory <dir>

Use <dir> to store the resulting files, instead of the current working directory.

-n , --numbered

Name output in *[PATCH n/m]* format, even with a single patch.

-N , --no-numbered

Name output in *[PATCH]* format.

--start-number <n>

Start numbering the patches at <n> instead of 1.

--numbered-files

Output file names will be a simple number sequence without the default first line of the commit appended.

-k , --keep-subject

Do not strip/add *[PATCH]* from the first line of the commit log message.

-s , --signoff

Add a *Signed-off-by* trailer to the commit message, using the committer identity of yourself. See the signoff option in Section G.3.28, “git-commit(1)” for more information.

--stdout

Print all commits to the standard output in mbox format, instead of creating a file for each one.

--attach[=<boundary>]

Create multipart/mixed attachment, the first part of which is the commit message and the patch itself in the second part, with *Content-Disposition: attachment*.

--no-attach

Disable the creation of an attachment, overriding the configuration setting.

--inline[=<boundary>]

Create multipart/mixed attachment, the first part of which is the commit message and the patch itself in the second part, with *Content-Disposition: inline*.

--thread[=<style>] , --no-thread

Controls addition of *In-Reply-To* and *References* headers to make the second and subsequent mails appear as replies to the first. Also controls generation of the *Message-ID* header to reference.

The optional <style> argument can be either *shallow* or *deep*. *shallow* threading makes every mail a reply to the head of the series, where the head is chosen from the cover letter, the *--in-reply-to*, and the first patch mail, in this order. *deep* threading makes every mail a reply to the previous one.

The default is *--no-thread*, unless the *format.thread* configuration is set. *--thread* without an argument is equivalent to *--thread=shallow*.

Beware that the default for *git send-email* is to thread emails itself. If you want *git format-patch* to take care of threading, you will want to ensure that threading is disabled for *git send-email*.

--in-reply-to=<message-id>

Make the first mail (or all the mails with *--no-thread*) appear as a reply to the given <message-id>, which avoids breaking threads to provide a new patch series.

--ignore-if-in-upstream

Do not include a patch that matches a commit in <until>..<since>. This will examine all patches reachable from <since> but not from <until> and compare them with the patches being generated, and any patch that matches is ignored.

--always

Include patches for commits that do not introduce any change, which are omitted by default.

--cover-from-description=<mode>

Controls which parts of the cover letter will be automatically populated using the branch's description.

If *<mode>* is *message* or *default*, the cover letter subject will be populated with placeholder text. The body of the cover letter will be populated with the branch's description. This is the default mode when no configuration nor command line option is specified.

If *<mode>* is *subject*, the first paragraph of the branch description will populate the cover letter subject. The remainder of the description will populate the body of the cover letter.

If *<mode>* is *auto*, if the first paragraph of the branch description is greater than 100 bytes, then the mode will be *message*, otherwise *subject* will be used.

If *<mode>* is *none*, both the cover letter subject and body will be populated with placeholder text.

--description-file=<file>

Use the contents of <file> instead of the branch's description for generating the cover letter.

--subject-prefix=<subject-prefix>

Instead of the standard *[PATCH]* prefix in the subject line, instead use *[<subject-prefix>]*. This can be used to name a patch series, and can be combined with the *--numbered* option.

The configuration variable *format.subjectPrefix* may also be used to configure a subject prefix to apply to a given repository for all patches. This is often useful on mailing lists which receive patches for several repositories and can be used to disambiguate the patches (with a value of e.g. "PATCH my-project").

--filename-max-length=<n>

Instead of the standard 64 bytes, chomp the generated output filenames at around *<n>* bytes (too short a value will be silently raised to a reasonable length). Defaults to the value of the *format.filenameMaxLength* configuration variable, or 64 if unconfigured.

--rfc

Prepends "RFC" to the subject prefix (producing "RFC PATCH" by default). RFC means "Request For Comments"; use this when sending an experimental patch for discussion rather than application.

-v <n> , --reroll-count=<n>

Mark the series as the <n>-th iteration of the topic. The output filenames have *v<n>* prepended to them, and the subject prefix ("PATCH" by default, but configurable via the *--subject-prefix* option) has `v<n>` appended to it. E.g. *--reroll-count=4* may produce *v4-0001-add-makefile.patch* file that has "Subject: [PATCH v4 1/20] Add makefile" in it. *<n>* does not have to be an integer (e.g. "--reroll-count=4.4", or "--reroll-count=4rev2" are allowed), but the downside of using such a reroll-count is that the range-diff/interdiff with the previous version does not state exactly which version the new iteration is compared against.

--to=<email>

Add a *To:* header to the email headers. This is in addition to any configured headers, and may be used multiple times. The negated form *--no-to* discards all *To:* headers added so far (from config or command line).

--cc=<email>

Add a *Cc:* header to the email headers. This is in addition to any configured headers, and may be used multiple times. The negated form *--no-cc* discards all *Cc:* headers added so far (from config or command line).

--from , --from=<ident>

Use *ident* in the *From:* header of each commit email. If the author ident of the commit is not textually identical to the provided *ident*, place a *From:* header in the body of the message with the original author. If no *ident* is given, use the committer ident.

Note that this option is only useful if you are actually sending the emails and want to identify yourself as the sender, but retain the original author (and *git am* will correctly pick up the in-body header). Note also that *git send-email* already handles this transformation for you, and this option should not be used if you are feeding the result to *git send-email*.

--[no-]force-in-body-from

With the e-mail sender specified via the *--from* option, by default, an in-body "From:" to identify the real author of the commit is added at the top of the commit log message if the sender is different from the author. With this option, the in-body "From:" is added even when the sender and the author have the same name and address, which may help if the mailing list software mangles the sender's identity. Defaults to the value of the *format.forceInBodyFrom* configuration variable.

--add-header=<header>

Add an arbitrary header to the email headers. This is in addition to any configured headers, and may be used multiple times. For example, *--add-header="Organization: git-foo"*. The negated form *--no-add-header* discards **all** (*To:*, *Cc:*, and custom) headers added so far from config or command line.

--[no-]cover-letter

In addition to the patches, generate a cover letter file containing the branch description, shortlog and the overall diffstat. You can fill in a description in the file before sending it out.

--encode-email-headers , --no-encode-email-headers

Encode email headers that have non-ASCII characters with "Q-encoding" (described in RFC 2047), instead of outputting the headers verbatim. Defaults to the value of the *format.encodeEmailHeaders* configuration variable.

--interdiff=<previous>

As a reviewer aid, insert an interdiff into the cover letter, or as commentary of the lone patch of a 1-patch series, showing the differences between the previous version of the patch series and the series currently being formatted. *previous* is a single revision naming the tip of the previous series which shares a common base with the series being formatted (for example *git format-patch --cover-letter --interdiff=feature/v1 -3 feature/v2*).

--range-diff=<previous>

As a reviewer aid, insert a range-diff (see Section G.3.105, “git-range-diff(1)”) into the cover letter, or as commentary of the lone patch of a 1-patch series, showing the differences between the previous version of the patch series and the series currently being formatted. *previous* can be a single revision naming the tip of the previous series if it shares a common base with the series being formatted (for example *git format-patch --cover-letter --range-diff=feature/v1 -3 feature/v2*), or a revision range if the two versions of the series are disjoint (for example *git format-patch --cover-letter --range-diff=feature/v1~3..feature/v1 -3 feature/v2*).

Note that diff options passed to the command affect how the primary product of *format-patch* is generated, and they are not passed to the underlying *range-diff* machinery used to generate the cover-letter material (this may change in the future).

--creation-factor=<percent>

Used with *--range-diff*, tweak the heuristic which matches up commits between the previous and current series of patches by adjusting the creation/deletion cost fudge factor. See Section G.3.105, “git-range-diff(1)”) for details.

--notes[=<ref>] , --no-notes

Append the notes (see Section G.3.94, “git-notes(1)”) for the commit after the three-dash line.

The expected use case of this is to write supporting explanation for the commit that does not belong to the commit log message proper, and include it with the patch submission. While one can simply write these explanations after *format-patch* has run but before sending, keeping them as Git notes allows them to be maintained between versions of the patch series (but see the discussion of the *notes.rewrite* configuration options in Section G.3.94, “git-notes(1)” to use this workflow).

The default is *--no-notes*, unless the *format.notes* configuration is set.

--[no-]signature=<signature>

Add a signature to each message produced. Per RFC 3676 the signature is separated from the body by a line with '-- ' on it. If the signature option is omitted the signature defaults to the Git version number.

--signature-file=<file>

Works just like --signature except the signature is read from a file.

--suffix=.<sfx>

Instead of using *.patch* as the suffix for generated filenames, use specified suffix. A common alternative is *--suffix=.txt*. Leaving this empty will remove the *.patch* suffix.

Note that the leading character does not have to be a dot; for example, you can use *--suffix=-patch* to get *0001-description-of-my-change-patch*.

-q , --quiet

Do not print the names of the generated files to standard output.

--no-binary

Do not output contents of changes in binary files, instead display a notice that those files changed. Patches generated using this option cannot be applied properly, but they are still useful for code review.

--zero-commit

Output an all-zero hash in each patch's From header instead of the hash of the commit.

--[no-]base[=<commit>]

Record the base tree information to identify the state the patch series applies to. See the BASE TREE INFORMATION section below for details. If <commit> is "auto", a base commit is automatically chosen. The *--no-base* option overrides a *format.useAutoBase* configuration.

--root

Treat the revision argument as a <revision-range>, even if it is just a single commit (that would normally be treated as a <since>). Note that root commits included in the specified range are always formatted as creation patches, independently of this flag.

--progress

Show progress reports on stderr as patches are generated.

CONFIGURATION

You can specify extra mail header lines to be added to each message, defaults for the subject prefix and file suffix, number patches when outputting more than one patch, add "To:" or "Cc:" headers, configure attachments, change the patch output directory, and sign off patches with configuration variables.

```
[format]
        headers = "Organization: git-foo\n"
        subjectPrefix = CHANGE
        suffix = .txt
        numbered = auto
        to = <email>
        cc = <email>
        attach [ = mime-boundary-string ]
        signOff = true
        outputDirectory = <directory>
        coverLetter = auto
        coverFromDescription = auto
```

DISCUSSION

The patch produced by *git format-patch* is in UNIX mailbox format, with a fixed "magic" time stamp to indicate that the file is output from format-patch rather than a real mailbox, like so:

```
From 8f72bad1baf19a53459661343e21d6491c3908d3 Mon Sep 17 00:00:00 2001
From: Tony Luck <tony.luck@intel.com>
Date: Tue, 13 Jul 2010 11:42:54 -0700
Subject: [PATCH] =?UTF-8?q?
[IA64]=20Put=20ia64=20config=20files=20on=20the=20?=
 =?UTF-8?q?Uwe=20Kleine-K=C3=B6nig=20diet?=
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 8bit

arch/arm config files were slimmed down using a python script
(See commit c2330e286f68f1c408b4aa6515ba49d57f05beae comment)

Do the same for ia64 so we can have sleek & trim looking
...
```

Typically it will be placed in a MUA's drafts folder, edited to add timely commentary that should not go in the changelog after the three dashes, and then sent as a message whose body, in our example, starts with "arch/arm config files were…". On the receiving end, readers can save interesting patches in a UNIX mailbox and apply them with Section G.3.3, “git-am(1)”.

When a patch is part of an ongoing discussion, the patch generated by *git format-patch* can be tweaked to take advantage of the *git am --scissors* feature. After your response to the discussion comes a line that consists solely of "-- >*8* --" (scissors and perforation), followed by the patch with unnecessary header fields removed:

```
...
> So we should do such-and-such.

Makes sense to me.  How about this patch?

-- >8 --
Subject: [IA64] Put ia64 config files on the Uwe Kleine-König diet
```

```
arch/arm config files were slimmed down using a python script
...
```

When sending a patch this way, most often you are sending your own patch, so in addition to the "*From $SHA1 $magic_timestamp*" marker you should omit *From:* and *Date:* lines from the patch file. The patch title is likely to be different from the subject of the discussion the patch is in response to, so it is likely that you would want to keep the Subject: line, like the example above.

1. Checking for patch corruption

Many mailers if not set up properly will corrupt whitespace. Here are two common types of corruption:

- Empty context lines that do not have *any* whitespace.
- Non-empty context lines that have one extra whitespace at the beginning.

One way to test if your MUA is set up correctly is:

- Send the patch to yourself, exactly the way you would, except with To: and Cc: lines that do not contain the list and maintainer address.
- Save that patch to a file in UNIX mailbox format. Call it a.patch, say.
- Apply it:

  ```
  $ git fetch <project> master:test-apply
  $ git switch test-apply
  $ git restore --source=HEAD --staged --worktree :/
  $ git am a.patch
  ```

If it does not apply correctly, there can be various reasons.

- The patch itself does not apply cleanly. That is *bad* but does not have much to do with your MUA. You might want to rebase the patch with Section G.3.107, “git-rebase(1)” before regenerating it in this case.
- The MUA corrupted your patch; "am" would complain that the patch does not apply. Look in the .git/rebase-apply/ subdirectory and see what *patch* file contains and check for the common corruption patterns mentioned above.
- While at it, check the *info* and *final-commit* files as well. If what is in *final-commit* is not exactly what you would want to see in the commit log message, it is very likely that the receiver would end up hand editing the log message when applying your patch. Things like "Hi, this is my first patch.\n" in the patch e-mail should come after the three-dash line that signals the end of the commit message.

MUA-SPECIFIC HINTS

Here are some hints on how to successfully submit patches inline using various mailers.

1. GMail

GMail does not have any way to turn off line wrapping in the web interface, so it will mangle any emails that you send. You can however use "git send-email" and send your patches through the GMail SMTP server, or use any IMAP email client to connect to the google IMAP server and forward the emails through that.

For hints on using *git send-email* to send your patches through the GMail SMTP server, see the EXAMPLE section of Section G.3.124, “git-send-email(1)”.

For hints on submission using the IMAP interface, see the EXAMPLE section of Section G.3.68, “git-imap-send(1)”.

2. Thunderbird

By default, Thunderbird will both wrap emails as well as flag them as being *format=flowed*, both of which will make the resulting email unusable by Git.

There are three different approaches: use an add-on to turn off line wraps, configure Thunderbird to not mangle patches, or use an external editor to keep Thunderbird from mangling the patches.

2.1. Approach #1 (add-on)

Install the Toggle Word Wrap add-on that is available from https://addons.mozilla.org/thunderbird/addon/toggle-word-wrap/ It adds a menu entry "Enable Word Wrap" in the composer's "Options" menu that you can tick off. Now you can compose the message as you otherwise do (cut + paste, *git format-patch | git imap-send*, etc), but you have to insert line breaks manually in any text that you type.

2.2. Approach #2 (configuration)

Three steps:

1. Configure your mail server composition as plain text: Edit…Account Settings…Composition & Addressing, uncheck "Compose Messages in HTML".

2. Configure your general composition window to not wrap.

   In Thunderbird 2: Edit..Preferences..Composition, wrap plain text messages at 0

   In Thunderbird 3: Edit..Preferences..Advanced..Config Editor. Search for "mail.wrap_long_lines". Toggle it to make sure it is set to *false*. Also, search for "mailnews.wraplength" and set the value to 0.

3. Disable the use of format=flowed: Edit..Preferences..Advanced..Config Editor. Search for "mailnews.send_plaintext_flowed". Toggle it to make sure it is set to *false*.

After that is done, you should be able to compose email as you otherwise would (cut + paste, *git format-patch | git imap-send*, etc), and the patches will not be mangled.

2.3. Approach #3 (external editor)

The following Thunderbird extensions are needed: AboutConfig from https://mjg.github.io/AboutConfig/ and External Editor from https://globs.org/articles.php?lng=en&pg=8

1. Prepare the patch as a text file using your method of choice.

2. Before opening a compose window, use Edit#Account Settings to uncheck the "Compose messages in HTML format" setting in the "Composition & Addressing" panel of the account to be used to send the patch.

3. In the main Thunderbird window, *before* you open the compose window for the patch, use Tools#about:config to set the following to the indicated values:

   ```
   mailnews.send_plaintext_flowed  => false
   mailnews.wraplength             => 0
   ```

4. Open a compose window and click the external editor icon.

5. In the external editor window, read in the patch file and exit the editor normally.

Side note: it may be possible to do step 2 with about:config and the following settings but no one's tried yet.

```
mail.html_compose                       => false
mail.identity.default.compose_html      => false
mail.identity.id?.compose_html          => false
```

There is a script in contrib/thunderbird-patch-inline which can help you include patches with Thunderbird in an easy way. To use it, do the steps above and then use the script as the external editor.

3. KMail

This should help you to submit patches inline using KMail.

1. Prepare the patch as a text file.
2. Click on New Mail.
3. Go under "Options" in the Composer window and be sure that "Word wrap" is not set.
4. Use Message → Insert file… and insert the patch.
5. Back in the compose window: add whatever other text you wish to the message, complete the addressing and subject fields, and press send.

BASE TREE INFORMATION

The base tree information block is used for maintainers or third party testers to know the exact state the patch series applies to. It consists of the *base commit*, which is a well-known commit that is part of the stable part of the project history everybody else works off of, and zero or more *prerequisite patches*, which are well-known patches in flight that is not yet part of the *base commit* that need to be applied on top of *base commit* in topological order before the patches can be applied.

The *base commit* is shown as "base-commit: " followed by the 40-hex of the commit object name. A *prerequisite patch* is shown as "prerequisite-patch-id: " followed by the 40-hex *patch id*, which can be obtained by passing the patch through the *git patch-id --stable* command.

Imagine that on top of the public commit P, you applied well-known patches X, Y and Z from somebody else, and then built your three-patch series A, B, C, the history would be like:

```
---P---X---Y---Z---A---B---C
```

With *git format-patch --base=P -3 C* (or variants thereof, e.g. with *--cover-letter* or using *Z..C* instead of *-3 C* to specify the range), the base tree information block is shown at the end of the first message the command outputs (either the first patch, or the cover letter), like this:

```
base-commit: P
prerequisite-patch-id: X
prerequisite-patch-id: Y
prerequisite-patch-id: Z
```

For non-linear topology, such as

```
---P---X---A---M---C
    \         /
     Y---Z---B
```

You can also use *git format-patch --base=P -3 C* to generate patches for A, B and C, and the identifiers for P, X, Y, Z are appended at the end of the first message.

If set *--base=auto* in cmdline, it will automatically compute the base commit as the merge base of tip commit of the remote-tracking branch and revision-range specified in cmdline. For a local branch, you need to make it to track a remote branch by *git branch --set-upstream-to* before using this option.

EXAMPLES

- Extract commits between revisions R1 and R2, and apply them on top of the current branch using *git am* to cherry-pick them:

```
$ git format-patch -k --stdout R1..R2 | git am -3 -k
```

- Extract all commits which are in the current branch but not in the origin branch:

  ```
  $ git format-patch origin
  ```

  For each commit a separate file is created in the current directory.

- Extract all commits that lead to *origin* since the inception of the project:

  ```
  $ git format-patch --root origin
  ```

- The same as the previous one:

  ```
  $ git format-patch -M -B origin
  ```

  Additionally, it detects and handles renames and complete rewrites intelligently to produce a renaming patch. A renaming patch reduces the amount of text output, and generally makes it easier to review. Note that non-Git "patch" programs won't understand renaming patches, so use it only when you know the recipient uses Git to apply your patch.

- Extract three topmost commits from the current branch and format them as e-mailable patches:

  ```
  $ git format-patch -3
  ```

CAVEATS

Note that *format-patch* will omit merge commits from the output, even if they are part of the requested range. A simple "patch" does not include enough information for the receiving end to reproduce the same merge commit.

SEE ALSO

Section G.3.3, “git-am(1)”, Section G.3.124, “git-send-email(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.55. git-fsck-objects(1)

2024-07-02
Revision History

2024-07-02

NAME

git-fsck-objects - Verifies the connectivity and validity of the objects in the database

SYNOPSIS

git fsck-objects …

DESCRIPTION

This is a synonym for Section G.3.56, “git-fsck(1)”. Please refer to the documentation of that command.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.56. git-fsck(1)

2024-07-02
Revision History

2024-07-02

NAME

git-fsck - Verifies the connectivity and validity of the objects in the database

SYNOPSIS

git fsck [--tags] [--root] [--unreachable] [--cache] [--no-reflogs]
[--[no-]full] [--strict] [--verbose] [--lost-found]
[--[no-]dangling] [--[no-]progress] [--connectivity-only]
[--[no-]name-objects] [<object>…]

DESCRIPTION

Verifies the connectivity and validity of the objects in the database.

OPTIONS

<object>

An object to treat as the head of an unreachability trace.

If no objects are given, *git fsck* defaults to using the index file, all SHA-1 references in the *refs* namespace, and all reflogs (unless --no-reflogs is given) as heads.

--unreachable

Print out objects that exist but that aren't reachable from any of the reference nodes.

--[no-]dangling

Print objects that exist but that are never *directly* used (default). *--no-dangling* can be used to omit this information from the output.

--root

Report root nodes.

--tags

Report tags.

--cache

Consider any object recorded in the index also as a head node for an unreachability trace.

--no-reflogs

Do not consider commits that are referenced only by an entry in a reflog to be reachable. This option is meant only to search for commits that used to be in a ref, but now aren't, but are still in that corresponding reflog.

--full

Check not just objects in GIT_OBJECT_DIRECTORY ($GIT_DIR/objects), but also the ones found in alternate object pools listed in GIT_ALTERNATE_OBJECT_DIRECTORIES or $GIT_DIR/objects/info/alternates, and in packed Git archives found in $GIT_DIR/objects/pack and corresponding pack subdirectories in alternate object pools. This is now default; you can turn it off with --no-full.

--connectivity-only

Check only the connectivity of reachable objects, making sure that any objects referenced by a reachable tag, commit, or tree are present. This speeds up the operation by avoiding reading blobs entirely (though it

does still check that referenced blobs exist). This will detect corruption in commits and trees, but not do any semantic checks (e.g., for format errors). Corruption in blob objects will not be detected at all.

Unreachable tags, commits, and trees will also be accessed to find the tips of dangling segments of history. Use *--no-dangling* if you don't care about this output and want to speed it up further.

--strict

Enable more strict checking, namely to catch a file mode recorded with g+w bit set, which was created by older versions of Git. Existing repositories, including the Linux kernel, Git itself, and sparse repository have old objects that trigger this check, but it is recommended to check new projects with this flag.

--verbose

Be chatty.

--lost-found

Write dangling objects into .git/lost-found/commit/ or .git/lost-found/other/, depending on type. If the object is a blob, the contents are written into the file, rather than its object name.

--name-objects

When displaying names of reachable objects, in addition to the SHA-1 also display a name that describes **how** they are reachable, compatible with Section G.3.121, “git-rev-parse(1)”, e.g. *HEAD@{1234567890}~25^2:s-rc/*.

--[no-]progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless --no-progress or --verbose is specified. --progress forces progress status even if the standard error stream is not directed to a terminal.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

fsck.<msg-id>

During fsck git may find issues with legacy data which wouldn't be generated by current versions of git, and which wouldn't be sent over the wire if *transfer.fsckObjects* was set. This feature is intended to support working with legacy repositories containing such data.

Setting *fsck.<msg-id>* will be picked up by Section G.3.56, “git-fsck(1)”, but to accept pushes of such data set *receive.fsck.<msg-id>* instead, or to clone or fetch it set *fetch.fsck.<msg-id>*.

The rest of the documentation discusses *fsck.** for brevity, but the same applies for the corresponding *receive.fsck.** and *fetch.fsck.**. variables.

Unlike variables like *color.ui* and *core.editor*, the *receive.fsck.<msg-id>* and *fetch.fsck.<msg-id>* variables will not fall back on the *fsck.<msg-id>* configuration if they aren't set. To uniformly configure the same fsck settings in different circumstances, all three of them must be set to the same values.

When *fsck.<msg-id>* is set, errors can be switched to warnings and vice versa by configuring the *fsck.<msg-id>* setting where the *<msg-id>* is the fsck message ID and the value is one of *error*, *warn* or *ignore*. For convenience, fsck prefixes the error/warning with the message ID, e.g. "missingEmail: invalid author/committer line - missing email" means that setting *fsck.missingEmail = ignore* will hide that issue.

In general, it is better to enumerate existing objects with problems with *fsck.skipList*, instead of listing the kind of breakages these problematic objects share to be ignored, as doing the latter will allow new instances of the same breakages go unnoticed.

Setting an unknown *fsck.<msg-id>* value will cause fsck to die, but doing the same for *receive.fsck.<msg-id>* and *fetch.fsck.<msg-id>* will only cause git to warn.

See the *Fsck Messages* section of Section G.3.56, “git-fsck(1)” for supported values of *<msg-id>*.

fsck.skipList

The path to a list of object names (i.e. one unabbreviated SHA-1 per line) that are known to be broken in a non-fatal way and should be ignored. On versions of Git 2.20 and later, comments (#), empty lines, and any leading and trailing whitespace are ignored. Everything but a SHA-1 per line will error out on older versions.

This feature is useful when an established project should be accepted despite early commits containing errors that can be safely ignored, such as invalid committer email addresses. Note: corrupt objects cannot be skipped with this setting.

Like *fsck.<msg-id>* this variable has corresponding *receive.fsck.skipList* and *fetch.fsck.skipList* variants.

Unlike variables like *color.ui* and *core.editor* the *receive.fsck.skipList* and *fetch.fsck.skipList* variables will not fall back on the *fsck.skipList* configuration if they aren't set. To uniformly configure the same fsck settings in different circumstances, all three of them must be set to the same values.

Older versions of Git (before 2.20) documented that the object names list should be sorted. This was never a requirement; the object names could appear in any order, but when reading the list we tracked whether the list was sorted for the purposes of an internal binary search implementation, which could save itself some work with an already sorted list. Unless you had a humongous list there was no reason to go out of your way to pre-sort the list. After Git version 2.20 a hash implementation is used instead, so there's now no reason to pre-sort the list.

DISCUSSION

git-fsck tests SHA-1 and general object sanity, and it does full tracking of the resulting reachability and everything else. It prints out any corruption it finds (missing or bad objects), and if you use the *--unreachable* flag it will also print out objects that exist but that aren't reachable from any of the specified head nodes (or the default set, as mentioned above).

Any corrupt objects you will have to find in backups or other archives (i.e., you can just remove them and do an *rsync* with some other site in the hopes that somebody else has the object you have corrupted).

If core.commitGraph is true, the commit-graph file will also be inspected using *git commit-graph verify*. See Section G.3.26, “git-commit-graph(1)”.

Extracted Diagnostics

unreachable <type> <object>

The <type> object <object>, isn't actually referred to directly or indirectly in any of the trees or commits seen. This can mean that there's another root node that you're not specifying or that the tree is corrupt. If you haven't missed a root node then you might as well delete unreachable nodes since they can't be used.

missing <type> <object>

The <type> object <object>, is referred to but isn't present in the database.

dangling <type> <object>

The <type> object <object>, is present in the database but never *directly* used. A dangling commit could be a root node.

hash mismatch <object>

The database has an object whose hash doesn't match the object database value. This indicates a serious data integrity problem.

FSCK MESSAGES

The following lists the types of errors *git fsck* detects and what each error means, with their default severity. The severity of the error, other than those that are marked as "(FATAL)", can be tweaked by setting the corresponding *fsck.<msg-id>* configuration variable.

badDate

(ERROR) Invalid date format in an author/committer line.

badDateOverflow

(ERROR) Invalid date value in an author/committer line.

badEmail

(ERROR) Invalid email format in an author/committer line.

badFilemode

(INFO) A tree contains a bad filemode entry.

badName

(ERROR) An author/committer name is empty.

badObjectSha1

(ERROR) An object has a bad sha1.

badParentSha1

(ERROR) A commit object has a bad parent sha1.

badTagName

(INFO) A tag has an invalid format.

badTimezone

(ERROR) Found an invalid time zone in an author/committer line.

badTree

(ERROR) A tree cannot be parsed.

badTreeSha1

(ERROR) A tree has an invalid format.

badType

(ERROR) Found an invalid object type.

duplicateEntries

(ERROR) A tree contains duplicate file entries.

emptyName

(WARN) A path contains an empty name.

extraHeaderEntry

(IGNORE) Extra headers found after *tagger*.

fullPathname

(WARN) A path contains the full path starting with "/".

gitattributesBlob

(ERROR) A non-blob found at *.gitattributes*.

gitattributesLarge

(ERROR) The *.gitattributes* blob is too large.

gitattributesLineLength

(ERROR) The *.gitattributes* blob contains too long lines.

gitattributesMissing

(ERROR) Unable to read *.gitattributes* blob.

gitattributesSymlink

(INFO) *.gitattributes* is a symlink.

gitignoreSymlink

(INFO) *.gitignore* is a symlink.

gitmodulesBlob

(ERROR) A non-blob found at *.gitmodules*.

gitmodulesLarge

(ERROR) The *.gitmodules* file is too large to parse.

gitmodulesMissing

(ERROR) Unable to read *.gitmodules* blob.

gitmodulesName

(ERROR) A submodule name is invalid.

gitmodulesParse

(INFO) Could not parse *.gitmodules* blob.

gitmodulesLarge; (ERROR) *.gitmodules* blob is too large to parse.

gitmodulesPath

(ERROR) *.gitmodules* path is invalid.

gitmodulesSymlink

(ERROR) *.gitmodules* is a symlink.

gitmodulesUpdate

(ERROR) Found an invalid submodule update setting.

gitmodulesUrl

(ERROR) Found an invalid submodule url.

hasDot

(WARN) A tree contains an entry named ..

hasDotdot

(WARN) A tree contains an entry named ...

hasDotgit

(WARN) A tree contains an entry named *.git*.

largePathname

(WARN) A tree contains an entry with a very long path name. If the value of *fsck.largePathname* contains a colon, that value is used as the maximum allowable length (e.g., "warn:10" would complain about any path component of 11 or more bytes). The default value is 4096.

mailmapSymlink

(INFO) *.mailmap* is a symlink.

missingAuthor

(ERROR) Author is missing.

missingCommitter

(ERROR) Committer is missing.

missingEmail

(ERROR) Email is missing in an author/committer line.

missingNameBeforeEmail

(ERROR) Missing name before an email in an author/committer line.

missingObject

(ERROR) Missing *object* line in tag object.

missingSpaceBeforeDate

(ERROR) Missing space before date in an author/committer line.

missingSpaceBeforeEmail

(ERROR) Missing space before the email in an author/committer line.

missingTag

(ERROR) Unexpected end after *type* line in a tag object.

missingTagEntry

(ERROR) Missing *tag* line in a tag object.

missingTaggerEntry

(INFO) Missing *tagger* line in a tag object.

missingTree

(ERROR) Missing *tree* line in a commit object.

missingType

(ERROR) Invalid type value on the *type* line in a tag object.

missingTypeEntry

(ERROR) Missing *type* line in a tag object.

multipleAuthors

(ERROR) Multiple author lines found in a commit.

nulInCommit

(WARN) Found a NUL byte in the commit object body.

nulInHeader

(FATAL) NUL byte exists in the object header.

nullSha1

(WARN) Tree contains entries pointing to a null sha1.

treeNotSorted

(ERROR) A tree is not properly sorted.

unknownType

(ERROR) Found an unknown object type.

unterminatedHeader

(FATAL) Missing end-of-line in the object header.

zeroPaddedDate

(ERROR) Found a zero padded date in an author/committer line.

zeroPaddedFilemode

(WARN) Found a zero padded filemode in a tree.

Environment Variables

GIT_OBJECT_DIRECTORY

used to specify the object database root (usually $GIT_DIR/objects)

GIT_INDEX_FILE

used to specify the index file of the index

GIT_ALTERNATE_OBJECT_DIRECTORIES

used to specify additional object database roots (usually unset)

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.57. git-fsmonitor--daemon(1)

2024-07-02

Revision History

2024-07-02

NAME

git-fsmonitor--daemon - A Built-in Filesystem Monitor

SYNOPSIS

git fsmonitor--daemon start
git fsmonitor--daemon run
git fsmonitor--daemon stop
git fsmonitor--daemon status

DESCRIPTION

A daemon to watch the working directory for file and directory changes using platform-specific filesystem notification facilities.

This daemon communicates directly with commands like *git status* using the *simple IPC* [https://www.kernel.org/pub/software/scm/git/docs/technical/api-simple-ipc.html] interface instead of the slower Section G.4.7, “githooks(5)” interface.

This daemon is built into Git so that no third-party tools are required.

OPTIONS

start

Starts a daemon in the background.

run

Runs a daemon in the foreground.

stop

Stops the daemon running in the current working directory, if present.

status

Exits with zero status if a daemon is watching the current working directory.

REMARKS

This daemon is a long running process used to watch a single working directory and maintain a list of the recently changed files and directories. Performance of commands such as *git status* can be increased if they just ask for a summary of changes to the working directory and can avoid scanning the disk.

When *core.fsmonitor* is set to *true* (see Section G.3.29, “git-config(1)”) commands, such as *git status*, will ask the daemon for changes and automatically start it (if necessary).

For more information see the "File System Monitor" section in Section G.3.147, “git-update-index(1)”.

CAVEATS

The fsmonitor daemon does not currently know about submodules and does not know to filter out filesystem events that happen within a submodule. If fsmonitor daemon is watching a super repo and a file is modified within the working directory of a submodule, it will report the change (as happening against the super repo). However, the client will properly ignore these extra events, so performance may be affected but it will not cause an incorrect result.

By default, the fsmonitor daemon refuses to work with network-mounted repositories; this may be overridden by setting *fsmonitor.allowRemote* to *true*. Note, however, that the fsmonitor daemon is not guaranteed to work correctly with all network-mounted repositories, so such use is considered experimental.

On Mac OS, the inter-process communication (IPC) between various Git commands and the fsmonitor daemon is done via a Unix domain socket (UDS) -- a special type of file -- which is supported by native Mac OS filesystems, but not on network-mounted filesystems, NTFS, or FAT32. Other filesystems may or may not have the needed support; the fsmonitor daemon is not guaranteed to work with these filesystems and such use is considered experimental.

By default, the socket is created in the *.git* directory. However, if the *.git* directory is on a network-mounted filesystem, it will instead be created at *$HOME/.git-fsmonitor-** unless *$HOME* itself is on a network-mounted filesystem, in which case you must set the configuration variable *fsmonitor.socketDir* to the path of a directory on a Mac OS native filesystem in which to create the socket file.

If none of the above directories (*.git*, *$HOME*, or *fsmonitor.socketDir*) is on a native Mac OS file filesystem the fsmonitor daemon will report an error that will cause the daemon and the currently running command to exit.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

fsmonitor.allowRemote

By default, the fsmonitor daemon refuses to work with network-mounted repositories. Setting *fsmonitor.allowRemote* to *true* overrides this behavior. Only respected when *core.fsmonitor* is set to *true*.

fsmonitor.socketDir

This Mac OS-specific option, if set, specifies the directory in which to create the Unix domain socket used for communication between the fsmonitor daemon and various Git commands. The directory must reside on a native Mac OS filesystem. Only respected when *core.fsmonitor* is set to *true*.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.58. git-gc(1)

2024-07-02

Revision History

2024-07-02

NAME

git-gc - Cleanup unnecessary files and optimize the local repository

SYNOPSIS

git gc [--aggressive] [--auto] [--quiet] [--prune=<date> | --no-prune] [--force] [--keep-largest-pack]

DESCRIPTION

Runs a number of housekeeping tasks within the current repository, such as compressing file revisions (to reduce disk space and increase performance), removing unreachable objects which may have been created from prior invocations of *git add*, packing refs, pruning reflog, rerere metadata or stale working trees. May also update ancillary indexes such as the commit-graph.

When common porcelain operations that create objects are run, they will check whether the repository has grown substantially since the last maintenance, and if so run *git gc* automatically. See *gc.auto* below for how to disable this behavior.

Running *git gc* manually should only be needed when adding objects to a repository without regularly running such porcelain commands, to do a one-off repository optimization, or e.g. to clean up a suboptimal mass-import. See the "PACKFILE OPTIMIZATION" section in Section G.3.47, “git-fast-import(1)” for more details on the import case.

OPTIONS

--aggressive

Usually *git gc* runs very quickly while providing good disk space utilization and performance. This option will cause *git gc* to more aggressively optimize the repository at the expense of taking much more time. The effects of this optimization are mostly persistent. See the "AGGRESSIVE" section below for details.

--auto

With this option, *git gc* checks whether any housekeeping is required; if not, it exits without performing any work.

See the *gc.auto* option in the "CONFIGURATION" section below for how this heuristic works.

Once housekeeping is triggered by exceeding the limits of configuration options such as *gc.auto* and *gc.autoPackLimit*, all other housekeeping tasks (e.g. rerere, working trees, reflog…) will be performed as well.

--[no-]cruft

When expiring unreachable objects, pack them separately into a cruft pack instead of storing them as loose objects. *--cruft* is on by default.

--max-cruft-size=<n>

When packing unreachable objects into a cruft pack, limit the size of new cruft packs to be at most *<n>* bytes. Overrides any value specified via the *gc.maxCruftSize* configuration. See the *--max-cruft-size* option of Section G.3.113, “git-repack(1)” for more.

--prune=<date>

Prune loose objects older than date (default is 2 weeks ago, overridable by the config variable *gc.pruneExpire*). --prune=now prunes loose objects regardless of their age and increases the risk of corruption if another process is writing to the repository concurrently; see "NOTES" below. --prune is on by default.

--no-prune

Do not prune any loose objects.

--quiet

Suppress all progress reports.

--force

Force *git gc* to run even if there may be another *git gc* instance running on this repository.

--keep-largest-pack

All packs except the largest non-cruft pack, any packs marked with a *.keep* file, and any cruft pack(s) are consolidated into a single pack. When this option is used, *gc.bigPackThreshold* is ignored.

AGGRESSIVE

When the *--aggressive* option is supplied, Section G.3.113, “git-repack(1)” will be invoked with the *-f* flag, which in turn will pass *--no-reuse-delta* to Section G.3.96, “git-pack-objects(1)”. This will throw away any existing deltas and re-compute them, at the expense of spending much more time on the repacking.

The effects of this are mostly persistent, e.g. when packs and loose objects are coalesced into one another pack the existing deltas in that pack might get re-used, but there are also various cases where we might pick a sub-optimal delta from a newer pack instead.

Furthermore, supplying *--aggressive* will tweak the *--depth* and *--window* options passed to Section G.3.113, "git-repack(1)". See the *gc.aggressiveDepth* and *gc.aggressiveWindow* settings below. By using a larger window size we're more likely to find more optimal deltas.

It's probably not worth it to use this option on a given repository without running tailored performance benchmarks on it. It takes a lot more time, and the resulting space/delta optimization may or may not be worth it. Not using this at all is the right trade-off for most users and their repositories.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, "git-config(1)" documentation. The content is the same as what's found there:

gc.aggressiveDepth

The depth parameter used in the delta compression algorithm used by *git gc --aggressive*. This defaults to 50, which is the default for the *--depth* option when *--aggressive* isn't in use.

See the documentation for the *--depth* option in Section G.3.113, "git-repack(1)" for more details.

gc.aggressiveWindow

The window size parameter used in the delta compression algorithm used by *git gc --aggressive*. This defaults to 250, which is a much more aggressive window size than the default *--window* of 10.

See the documentation for the *--window* option in Section G.3.113, "git-repack(1)" for more details.

gc.auto

When there are approximately more than this many loose objects in the repository, *git gc --auto* will pack them. Some Porcelain commands use this command to perform a light-weight garbage collection from time to time. The default value is 6700.

Setting this to 0 disables not only automatic packing based on the number of loose objects, but also any other heuristic *git gc --auto* will otherwise use to determine if there's work to do, such as *gc.autoPackLimit*.

gc.autoPackLimit

When there are more than this many packs that are not marked with **.keep* file in the repository, *git gc --auto* consolidates them into one larger pack. The default value is 50. Setting this to 0 disables it. Setting *gc.auto* to 0 will also disable this.

See the *gc.bigPackThreshold* configuration variable below. When in use, it'll affect how the auto pack limit works.

gc.autoDetach

Make *git gc --auto* return immediately and run in the background if the system supports it. Default is true.

gc.bigPackThreshold

If non-zero, all non-cruft packs larger than this limit are kept when *git gc* is run. This is very similar to *--keep-largest-pack* except that all non-cruft packs that meet the threshold are kept, not just the largest pack. Defaults to zero. Common unit suffixes of *k*, *m*, or *g* are supported.

Note that if the number of kept packs is more than gc.autoPackLimit, this configuration variable is ignored, all packs except the base pack will be repacked. After this the number of packs should go below gc.autoPackLimit and gc.bigPackThreshold should be respected again.

If the amount of memory estimated for *git repack* to run smoothly is not available and *gc.bigPackThreshold* is not set, the largest pack will also be excluded (this is the equivalent of running *git gc* with *--keep-largest-pack*).

gc.writeCommitGraph

If true, then gc will rewrite the commit-graph file when Section G.3.58, “git-gc(1)” is run. When using *git gc --auto* the commit-graph will be updated if housekeeping is required. Default is true. See Section G.3.26, “git-commit-graph(1)” for details.

gc.logExpiry

If the file gc.log exists, then *git gc --auto* will print its content and exit with status zero instead of running unless that file is more than *gc.logExpiry* old. Default is "1.day". See *gc.pruneExpire* for more ways to specify its value.

gc.packRefs

Running *git pack-refs* in a repository renders it unclonable by Git versions prior to 1.5.1.2 over dumb transports such as HTTP. This variable determines whether *git gc* runs *git pack-refs*. This can be set to *notbare* to enable it within all non-bare repos or it can be set to a boolean value. The default is *true*.

gc.cruftPacks

Store unreachable objects in a cruft pack (see Section G.3.113, “git-repack(1)”) instead of as loose objects. The default is *true*.

gc.maxCruftSize

Limit the size of new cruft packs when repacking. When specified in addition to *--max-cruft-size*, the command line option takes priority. See the *--max-cruft-size* option of Section G.3.113, “git-repack(1)”.

gc.pruneExpire

When *git gc* is run, it will call *prune --expire 2.weeks.ago* (and *repack --cruft --cruft-expiration 2.weeks.ago* if using cruft packs via *gc.cruftPacks* or *--cruft*). Override the grace period with this config variable. The value "now" may be used to disable this grace period and always prune unreachable objects immediately, or "never" may be used to suppress pruning. This feature helps prevent corruption when *git gc* runs concurrently with another process writing to the repository; see the "NOTES" section of Section G.3.58, “git-gc(1)”.

gc.worktreePruneExpire

When *git gc* is run, it calls *git worktree prune --expire 3.months.ago*. This config variable can be used to set a different grace period. The value "now" may be used to disable the grace period and prune *$GIT_DIR/worktrees* immediately, or "never" may be used to suppress pruning.

gc.reflogExpire , gc.<pattern>.reflogExpire

git reflog expire removes reflog entries older than this time; defaults to 90 days. The value "now" expires all entries immediately, and "never" suppresses expiration altogether. With "<pattern>" (e.g. "refs/stash") in the middle the setting applies only to the refs that match the <pattern>.

gc.reflogExpireUnreachable , gc.<pattern>.reflogExpireUnreachable

git reflog expire removes reflog entries older than this time and are not reachable from the current tip; defaults to 30 days. The value "now" expires all entries immediately, and "never" suppresses expiration altogether. With "<pattern>" (e.g. "refs/stash") in the middle, the setting applies only to the refs that match the <pattern>.

These types of entries are generally created as a result of using *git commit --amend* or *git rebase* and are the commits prior to the amend or rebase occurring. Since these changes are not part of the current project most users will want to expire them sooner, which is why the default is more aggressive than *gc.reflogExpire*.

gc.recentObjectsHook

When considering whether or not to remove an object (either when generating a cruft pack or storing unreachable objects as loose), use the shell to execute the specified command(s). Interpret their output as object IDs which Git will consider as "recent", regardless of their age. By treating their mtimes as "now", any objects (and their descendants) mentioned in the output will be kept regardless of their true age.

Output must contain exactly one hex object ID per line, and nothing else. Objects which cannot be found in the repository are ignored. Multiple hooks are supported, but all must exit successfully, else the operation (either generating a cruft pack or unpacking unreachable objects) will be halted.

gc.repackFilter

When repacking, use the specified filter to move certain objects into a separate packfile. See the *--filter=<filter-spec>* option of Section G.3.113, “git-repack(1)”.

gc.repackFilterTo

When repacking and using a filter, see *gc.repackFilter*, the specified location will be used to create the packfile containing the filtered out objects. **WARNING:** The specified location should be accessible, using for example the Git alternates mechanism, otherwise the repo could be considered corrupt by Git as it migh not be able to access the objects in that packfile. See the *--filter-to=<dir>* option of Section G.3.113, “git-repack(1)” and the *objects/info/alternates* section of Section G.4.13, “gitrepository-layout(5)”.

gc.rerereResolved

Records of conflicted merge you resolved earlier are kept for this many days when *git rerere gc* is run. You can also use more human-readable "1.month.ago", etc. The default is 60 days. See Section G.3.117, “git-rerere(1)”.

gc.rerereUnresolved

Records of conflicted merge you have not resolved are kept for this many days when *git rerere gc* is run. You can also use more human-readable "1.month.ago", etc. The default is 15 days. See Section G.3.117, “git-rerere(1)”.

NOTES

git gc tries very hard not to delete objects that are referenced anywhere in your repository. In particular, it will keep not only objects referenced by your current set of branches and tags, but also objects referenced by the index, remote-tracking branches, reflogs (which may reference commits in branches that were later amended or rewound), and anything else in the refs/* namespace. Note that a note (of the kind created by *git notes*) attached to an object does not contribute in keeping the object alive. If you are expecting some objects to be deleted and they aren't, check all of those locations and decide whether it makes sense in your case to remove those references.

On the other hand, when *git gc* runs concurrently with another process, there is a risk of it deleting an object that the other process is using but hasn't created a reference to. This may just cause the other process to fail or may corrupt the repository if the other process later adds a reference to the deleted object. Git has two features that significantly mitigate this problem:

1. Any object with modification time newer than the *--prune* date is kept, along with everything reachable from it.

2. Most operations that add an object to the database update the modification time of the object if it is already present so that #1 applies.

However, these features fall short of a complete solution, so users who run commands concurrently have to live with some risk of corruption (which seems to be low in practice).

HOOKS

The *git gc --auto* command will run the *pre-auto-gc* hook. See Section G.4.7, “githooks(5)” for more information.

SEE ALSO

Section G.3.101, “git-prune(1)” Section G.3.109, “git-reflog(1)” Section G.3.113, “git-repack(1)” Section G.3.117, “git-rerere(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.59. git-get-tar-commit-id(1)

2024-07-02
Revision History
2024-07-02

NAME

git-get-tar-commit-id - Extract commit ID from an archive created using git-archive

SYNOPSIS

git get-tar-commit-id

DESCRIPTION

Read a tar archive created by *git archive* from the standard input and extract the commit ID stored in it. It reads only the first 1024 bytes of input, thus its runtime is not influenced by the size of the tar archive very much.

If no commit ID is found, *git get-tar-commit-id* quietly exits with a return code of 1. This can happen if the archive had not been created using *git archive* or if the first parameter of *git archive* had been a tree ID instead of a commit ID or tag.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.60. git-grep(1)

2024-07-02
Revision History
2024-07-02

NAME

git-grep - Print lines matching a pattern

SYNOPSIS

git grep [-a | --text] [-I] [--textconv] [-i | --ignore-case] [-w | --word-regexp]
[-v | --invert-match] [-h|-H] [--full-name]
[-E | --extended-regexp] [-G | --basic-regexp]
[-P | --perl-regexp]
[-F | --fixed-strings] [-n | --line-number] [--column]
[-l | --files-with-matches] [-L | --files-without-match]
[(-O | --open-files-in-pager) [<pager>]]
[-z | --null]
[-o | --only-matching] [-c | --count] [--all-match] [-q | --quiet]
[--max-depth <depth>] [--[no-]recursive]
[--color[=<when>] | --no-color]
[--break] [--heading] [-p | --show-function]
[-A <post-context>] [-B <pre-context>] [-C <context>]
[-W | --function-context]
[(-m | --max-count) <num>]

[--threads <num>]
[-f <file>] [-e] <pattern>
[--and|--or|--not|(|)|-e <pattern>…]
[--recurse-submodules] [--parent-basename <basename>]
[[--[no-]exclude-standard] [--cached | --untracked | --no-index] | <tree>…]
[--] [<pathspec>…]

DESCRIPTION

Look for specified patterns in the tracked files in the work tree, blobs registered in the index file, or blobs in given tree objects. Patterns are lists of one or more search expressions separated by newline characters. An empty string as search expression matches all lines.

OPTIONS

--cached

Instead of searching tracked files in the working tree, search blobs registered in the index file.

--untracked

In addition to searching in the tracked files in the working tree, search also in untracked files.

--no-index

Search files in the current directory that is not managed by Git, or by ignoring that the current directory is managed by Git. This is rather similar to running the regular *grep(1)* utility with its *-r* option specified, but with some additional benefits, such as using pathspec patterns to limit paths; see the *pathspec* entry in Section G.4.19, “gitglossary(7)” for more information.

This option cannot be used together with *--cached* or *--untracked*. See also *grep.fallbackToNoIndex* in *CONFIGURATION* below.

--no-exclude-standard

Also search in ignored files by not honoring the *.gitignore* mechanism. Only useful with *--untracked*.

--exclude-standard

Do not pay attention to ignored files specified via the *.gitignore* mechanism. Only useful when searching files in the current directory with *--no-index*.

--recurse-submodules

Recursively search in each submodule that is active and checked out in the repository. When used in combination with the *<tree>* option the prefix of all submodule output will be the name of the parent project's *<tree>* object. This option cannot be used together with *--untracked*, and it has no effect if *--no-index* is specified.

-a , --text

Process binary files as if they were text.

--textconv

Honor textconv filter settings.

--no-textconv

Do not honor textconv filter settings. This is the default.

-i , --ignore-case

Ignore case differences between the patterns and the files.

-I

Don't match the pattern in binary files.

--max-depth <depth>

For each <pathspec> given on command line, descend at most <depth> levels of directories. A value of -1 means no limit. This option is ignored if <pathspec> contains active wildcards. In other words if "a*" matches a directory named "a*", "*" is matched literally so --max-depth is still effective.

-r , --recursive

Same as *--max-depth=-1*; this is the default.

--no-recursive

Same as *--max-depth=0*.

-w , --word-regexp

Match the pattern only at word boundary (either begin at the beginning of a line, or preceded by a non-word character; end at the end of a line or followed by a non-word character).

-v , --invert-match

Select non-matching lines.

-h , -H

By default, the command shows the filename for each match. *-h* option is used to suppress this output. *-H* is there for completeness and does not do anything except it overrides *-h* given earlier on the command line.

--full-name

When run from a subdirectory, the command usually outputs paths relative to the current directory. This option forces paths to be output relative to the project top directory.

-E , --extended-regexp , -G , --basic-regexp

Use POSIX extended/basic regexp for patterns. Default is to use basic regexp.

-P , --perl-regexp

Use Perl-compatible regular expressions for patterns.

Support for these types of regular expressions is an optional compile-time dependency. If Git wasn't compiled with support for them providing this option will cause it to die.

-F , --fixed-strings

Use fixed strings for patterns (don't interpret pattern as a regex).

-n , --line-number

Prefix the line number to matching lines.

--column

Prefix the 1-indexed byte-offset of the first match from the start of the matching line.

-l , --files-with-matches , --name-only , -L , --files-without-match

Instead of showing every matched line, show only the names of files that contain (or do not contain) matches. For better compatibility with *git diff*, *--name-only* is a synonym for *--files-with-matches*.

-O[<pager>] , --open-files-in-pager[=<pager>]

Open the matching files in the pager (not the output of *grep*). If the pager happens to be "less" or "vi", and the user specified only one pattern, the first file is positioned at the first match automatically. The *pager* argument is optional; if specified, it must be stuck to the option without a space. If *pager* is unspecified, the default pager will be used (see *core.pager* in Section G.3.29, “git-config(1)”).

-z , --null

Use \0 as the delimiter for pathnames in the output, and print them verbatim. Without this option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

-o , --only-matching

Print only the matched (non-empty) parts of a matching line, with each such part on a separate output line.

-c , --count

Instead of showing every matched line, show the number of lines that match.

--color[=<when>]

Show colored matches. The value must be always (the default), never, or auto.

--no-color

Turn off match highlighting, even when the configuration file gives the default to color output. Same as *--color=never*.

--break

Print an empty line between matches from different files.

--heading

Show the filename above the matches in that file instead of at the start of each shown line.

-p , --show-function

Show the preceding line that contains the function name of the match, unless the matching line is a function name itself. The name is determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

-<num> , -C <num> , --context <num>

Show <num> leading and trailing lines, and place a line containing -- between contiguous groups of matches.

-A <num> , --after-context <num>

Show <num> trailing lines, and place a line containing -- between contiguous groups of matches.

-B <num> , --before-context <num>

Show <num> leading lines, and place a line containing -- between contiguous groups of matches.

-W , --function-context

Show the surrounding text from the previous line containing a function name up to the one before the next function name, effectively showing the whole function in which the match was found. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

-m <num> , --max-count <num>

Limit the amount of matches per file. When using the *-v* or *--invert-match* option, the search stops after the specified number of non-matches. A value of -1 will return unlimited results (the default). A value of 0 will exit immediately with a non-zero status.

--threads <num>

Number of *grep* worker threads to use. See *NOTES ON THREADS* and *grep.threads* in *CONFIGURATION* for more information.

-f <file>

Read patterns from <file>, one per line.

Passing the pattern via <file> allows for providing a search pattern containing a \0.

Not all pattern types support patterns containing \0. Git will error out if a given pattern type can't support such a pattern. The *--perl-regexp* pattern type when compiled against the PCRE v2 backend has the widest support for these types of patterns.

In versions of Git before 2.23.0 patterns containing \0 would be silently considered fixed. This was never documented, there were also odd and undocumented interactions between e.g. non-ASCII patterns containing \0 and *--ignore-case*.

In future versions we may learn to support patterns containing \0 for more search backends, until then we'll die when the pattern type in question doesn't support them.

-e

The next parameter is the pattern. This option has to be used for patterns starting with - and should be used in scripts passing user input to grep. Multiple patterns are combined by *or*.

--and , --or , --not , (…)

Specify how multiple patterns are combined using Boolean expressions. *--or* is the default operator. *--and* has higher precedence than *--or*. *-e* has to be used for all patterns.

--all-match

When giving multiple pattern expressions combined with *--or*, this flag is specified to limit the match to files that have lines to match all of them.

-q , --quiet

Do not output matched lines; instead, exit with status 0 when there is a match and with non-zero status when there isn't.

<tree>…

Instead of searching tracked files in the working tree, search blobs in the given trees.

--

Signals the end of options; the rest of the parameters are <pathspec> limiters.

<pathspec>…

If given, limit the search to paths matching at least one pattern. Both leading paths match and glob(7) patterns are supported.

For more details about the <pathspec> syntax, see the *pathspec* entry in Section G.4.19, “gitglossary(7)”.

EXAMPLES

git grep 'time_t' -- '.[ch]'*

Looks for *time_t* in all tracked .c and .h files in the working directory and its subdirectories.

git grep -e '#define' --and \(-e MAX_PATH -e PATH_MAX \)

Looks for a line that has *#define* and either *MAX_PATH* or *PATH_MAX*.

git grep --all-match -e NODE -e Unexpected

Looks for a line that has *NODE* or *Unexpected* in files that have lines that match both.

git grep solution -- :^Documentation

Looks for *solution*, excluding files in *Documentation*.

NOTES ON THREADS

The *--threads* option (and the *grep.threads* configuration) will be ignored when *--open-files-in-pager* is used, forcing a single-threaded execution.

When grepping the object store (with *--cached* or giving tree objects), running with multiple threads might perform slower than single-threaded if *--textconv* is given and there are too many text conversions. Thus, if low performance is experienced in this case, it might be desirable to use *--threads=1*.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

grep.lineNumber

If set to true, enable *-n* option by default.

grep.column

If set to true, enable the *--column* option by default.

grep.patternType

Set the default matching behavior. Using a value of *basic*, *extended*, *fixed*, or *perl* will enable the *--basic-regexp*, *--extended-regexp*, *--fixed-strings*, or *--perl-regexp* option accordingly, while the value *default* will use the *grep.extendedRegexp* option to choose between *basic* and *extended*.

grep.extendedRegexp

If set to true, enable *--extended-regexp* option by default. This option is ignored when the *grep.patternType* option is set to a value other than *default*.

grep.threads

Number of grep worker threads to use. If unset (or set to 0), Git will use as many threads as the number of logical cores available.

grep.fullName

If set to true, enable *--full-name* option by default.

grep.fallbackToNoIndex

If set to true, fall back to *git grep --no-index* if *git grep* is executed outside of a git repository. Defaults to false.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.61. git-gui(1)

2024-07-02
Revision History
2024-07-02

NAME

git-gui - A portable graphical interface to Git

SYNOPSIS

git gui [<command>] [<arguments>]

DESCRIPTION

A Tcl/Tk based graphical user interface to Git. *git gui* focuses on allowing users to make changes to their repository by making new commits, amending existing ones, creating branches, performing local merges, and fetching/pushing to remote repositories.

Unlike *gitk*, *git gui* focuses on commit generation and single file annotation and does not show project history. It does however supply menu actions to start a *gitk* session from within *git gui*.

git gui is known to work on all popular UNIX systems, Mac OS X, and Windows (under both Cygwin and MSYS). To the extent possible OS specific user interface guidelines are followed, making *git gui* a fairly native interface for users.

COMMANDS

blame

Start a blame viewer on the specified file on the given version (or working directory if not specified).

browser

Start a tree browser showing all files in the specified commit. Files selected through the browser are opened in the blame viewer.

citool

Start *git gui* and arrange to make exactly one commit before exiting and returning to the shell. The interface is limited to only commit actions, slightly reducing the application's startup time and simplifying the menubar.

version

Display the currently running version of *git gui*.

Examples

git gui blame Makefile

Show the contents of the file *Makefile* in the current working directory, and provide annotations for both the original author of each line, and who moved the line to its current location. The uncommitted file is annotated, and uncommitted changes (if any) are explicitly attributed to *Not Yet Committed*.

git gui blame v0.99.8 Makefile

Show the contents of *Makefile* in revision *v0.99.8* and provide annotations for each line. Unlike the above example the file is read from the object database and not the working directory.

git gui blame --line=100 Makefile

Loads annotations as described above and automatically scrolls the view to center on line *100*.

git gui citool

Make one commit and return to the shell when it is complete. This command returns a non-zero exit code if the window was closed in any way other than by making a commit.

git gui citool --amend

Automatically enter the *Amend Last Commit* mode of the interface.

git gui citool --nocommit

Behave as normal citool, but instead of making a commit simply terminate with a zero exit code. It still checks that the index does not contain any unmerged entries, so you can use it as a GUI version of Section G.3.88, "git-mergetool(1)"

git citool

Same as *git gui citool* (above).

git gui browser maint

Show a browser for the tree of the *maint* branch. Files selected in the browser can be viewed with the internal blame viewer.

SEE ALSO

Section G.4.8, "gitk(1)"

The Git repository browser. Shows branches, commit history and file differences. gitk is the utility started by *git gui*'s Repository Visualize actions.

Other

git gui is actually maintained as an independent project, but stable versions are distributed as part of the Git suite for the convenience of end users.

The official repository of the *git gui* project can be found at:

`https://github.com/j6t/git-gui`

GIT

Part of the Section G.3.1, "git(1)" suite

G.3.62. git-hash-object(1)

2024-07-02
Revision History
2024-07-02

NAME

git-hash-object - Compute object ID and optionally create an object from a file

SYNOPSIS

git hash-object [-t <type>] [-w] [--path=<file> | --no-filters]
[--stdin [--literally]] [--] <file>…
git hash-object [-t <type>] [-w] --stdin-paths [--no-filters]

DESCRIPTION

Computes the object ID value for an object with specified type with the contents of the named file (which can be outside of the work tree), and optionally writes the resulting object into the object database. Reports its object ID to its standard output. When <type> is not specified, it defaults to "blob".

OPTIONS

-t <type>

Specify the type of object to be created (default: "blob"). Possible values are *commit*, *tree*, *blob*, and *tag*.

-w

Actually write the object into the object database.

--stdin

Read the object from standard input instead of from a file.

--stdin-paths

Read file names from the standard input, one per line, instead of from the command-line.

--path

Hash object as if it were located at the given path. The location of the file does not directly influence the hash value, but the path is used to determine which Git filters should be applied to the object before it can be placed in the object database. As a result of applying filters, the actual blob put into the object database may differ from the given file. This option is mainly useful for hashing temporary files located outside of the working directory or files read from stdin.

--no-filters

Hash the contents as is, ignoring any input filter that would have been chosen by the attributes mechanism, including the end-of-line conversion. If the file is read from standard input then this is always implied, unless the *--path* option is given.

--literally

Allow *--stdin* to hash any garbage into a loose object which might not otherwise pass standard object parsing or git-fsck checks. Useful for stress-testing Git itself or reproducing characteristics of corrupt or bogus objects encountered in the wild.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.63. git-help(1)

2024-07-02

Revision History

2024-07-02

NAME

git-help - Display help information about Git

SYNOPSIS

git help [-a|--all] [--[no-]verbose] [--[no-]external-commands] [--[no-]aliases]
git help [[-i|--info] [-m|--man] [-w|--web]] [<command>|<doc>]
git help [-g|--guides]
git help [-c|--config]

git help [--user-interfaces]
git help [--developer-interfaces]

DESCRIPTION

With no options and no *<command>* or *<doc>* given, the synopsis of the *git* command and a list of the most commonly used Git commands are printed on the standard output.

If the option *--all* or *-a* is given, all available commands are printed on the standard output.

If the option *--guides* or *-g* is given, a list of the Git concept guides is also printed on the standard output.

If a command or other documentation is given, the relevant manual page will be brought up. The *man* program is used by default for this purpose, but this can be overridden by other options or configuration variables.

If an alias is given, git shows the definition of the alias on standard output. To get the manual page for the aliased command, use *git <command> --help*.

Note that *git --help ...* is identical to *git help ...* because the former is internally converted into the latter.

To display the Section G.3.1, “git(1)” man page, use *git help git*.

This page can be displayed with *git help help* or *git help --help*.

OPTIONS

-a , --all

Print all the available commands on the standard output.

--no-external-commands

When used with *--all*, exclude the listing of external "git-*" commands found in the *$PATH*.

--no-aliases

When used with *--all*, exclude the listing of configured aliases.

--verbose

When used with *--all*, print description for all recognized commands. This is the default.

-c , --config

List all available configuration variables. This is a short summary of the list in Section G.3.29, “git-config(1)”.

-g , --guides

Print a list of the Git concept guides on the standard output.

--user-interfaces

Print a list of the repository, command and file interfaces documentation on the standard output.

In-repository file interfaces such as *.git/info/exclude* are documented here (see Section G.4.13, “gitrepository-layout(5)”), as well as in-tree configuration such as *.mailmap* (see Section G.4.9, “gitmailmap(5)”).

This section of the documentation also covers general or widespread user-interface conventions (e.g. Section G.4.1, “gitcli(7)”), and pseudo-configuration such as the file-based *.git/hooks/** interface described in Section G.4.7, “githooks(5)”.

--developer-interfaces

Print a list of file formats, protocols and other developer interfaces documentation on the standard output.

-i , --info

Display manual page for the command in the *info* format. The *info* program will be used for that purpose.

-m , --man

Display manual page for the command in the *man* format. This option may be used to override a value set in the *help.format* configuration variable.

By default the *man* program will be used to display the manual page, but the *man.viewer* configuration variable may be used to choose other display programs (see below).

-w , --web

Display manual page for the command in the *web* (HTML) format. A web browser will be used for that purpose.

The web browser can be specified using the configuration variable *help.browser*, or *web.browser* if the former is not set. If neither of these config variables is set, the *git web--browse* helper script (called by *git help*) will pick a suitable default. See Section G.3.157, “git-web--browse(1)” for more information about this.

CONFIGURATION VARIABLES

1. help.format

If no command-line option is passed, the *help.format* configuration variable will be checked. The following values are supported for this variable; they make *git help* behave as their corresponding command- line option:

- "man" corresponds to *-m*|*--man*,
- "info" corresponds to *-i*|*--info*,
- "web" or "html" correspond to *-w*|*--web*.

2. help.browser, web.browser, and browser.<tool>.path

The *help.browser*, *web.browser* and *browser.<tool>.path* will also be checked if the *web* format is chosen (either by command-line option or configuration variable). See *-w*|*--web* in the OPTIONS section above and Section G.3.157, “git-web--browse(1)”.

3. man.viewer

The *man.viewer* configuration variable will be checked if the *man* format is chosen. The following values are currently supported:

- "man": use the *man* program as usual,
- "woman": use *emacsclient* to launch the "woman" mode in emacs (this only works starting with emacsclient versions 22),
- "konqueror": use *kfmclient* to open the man page in a new konqueror tab (see *Note about konqueror* below).

Values for other tools can be used if there is a corresponding *man.<tool>.cmd* configuration entry (see below).

Multiple values may be given to the *man.viewer* configuration variable. Their corresponding programs will be tried in the order listed in the configuration file.

For example, this configuration:

```
[man]
        viewer = konqueror
```

```
                viewer = woman
```

will try to use konqueror first. But this may fail (for example, if DISPLAY is not set) and in that case emacs' woman mode will be tried.

If everything fails, or if no viewer is configured, the viewer specified in the *GIT_MAN_VIEWER* environment variable will be tried. If that fails too, the *man* program will be tried anyway.

4. man.<tool>.path

You can explicitly provide a full path to your preferred man viewer by setting the configuration variable *man.<tool>.path*. For example, you can configure the absolute path to konqueror by setting *man.konqueror.path*. Otherwise, *git help* assumes the tool is available in PATH.

5. man.<tool>.cmd

When the man viewer, specified by the *man.viewer* configuration variables, is not among the supported ones, then the corresponding *man.<tool>.cmd* configuration variable will be looked up. If this variable exists then the specified tool will be treated as a custom command and a shell eval will be used to run the command with the man page passed as arguments.

6. Note about konqueror

When *konqueror* is specified in the *man.viewer* configuration variable, we launch *kfmclient* to try to open the man page on an already opened konqueror in a new tab if possible.

For consistency, we also try such a trick if *man.konqueror.path* is set to something like *A_PATH_TO/konqueror*. That means we will try to launch *A_PATH_TO/kfmclient* instead.

If you really want to use *konqueror*, then you can use something like the following:

```
        [man]
                viewer = konq

        [man "konq"]
                cmd = A_PATH_TO/konqueror
```

7. Note about git config --global

Note that all these configuration variables should probably be set using the *--global* flag, for example like this:

```
$ git config --global help.format web
$ git config --global web.browser firefox
```

as they are probably more user specific than repository specific. See Section G.3.29, “git-config(1)” for more information about this.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.64. git-hook(1)

2024-07-02

Revision History

2024-07-02

NAME

git-hook - Run git hooks

SYNOPSIS

git hook run [--ignore-missing] [--to-stdin=<path>] <hook-name> [-- <hook-args>]

DESCRIPTION

A command interface for running git hooks (see Section G.4.7, “githooks(5)”), for use by other scripted git commands.

SUBCOMMANDS

run

Run the <*hook-name*> hook. See Section G.4.7, “githooks(5)” for supported hook names.

Any positional arguments to the hook should be passed after a mandatory -- (or *--end-of-options*, see Section G.4.1, “gitcli(7)”). See Section G.4.7, “githooks(5)” for arguments hooks might expect (if any).

OPTIONS

--to-stdin

For "run"; specify a file which will be streamed into the hook's stdin. The hook will receive the entire file from beginning to EOF.

--ignore-missing

Ignore any missing hook by quietly returning zero. Used for tools that want to do a blind one-shot run of a hook that may or may not be present.

SEE ALSO

Section G.4.7, “githooks(5)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.65. git-http-backend(1)

2024-07-02
Revision History
2024-07-02

NAME

git-http-backend - Server side implementation of Git over HTTP

SYNOPSIS

git http-backend

DESCRIPTION

A simple CGI program to serve the contents of a Git repository to Git clients accessing the repository over http:// and https:// protocols. The program supports clients fetching using both the smart HTTP protocol and the backwards-compatible dumb HTTP protocol, as well as clients pushing using the smart HTTP protocol. It also supports Git's more-efficient "v2" protocol if properly configured; see the discussion of *GIT_PROTOCOL* in the ENVIRONMENT section below.

It verifies that the directory has the magic file "git-daemon-export-ok", and it will refuse to export any Git directory that hasn't explicitly been marked for export this way (unless the *GIT_HTTP_EXPORT_ALL* environment variable is set).

By default, only the *upload-pack* service is enabled, which serves *git fetch-pack* and *git ls-remote* clients, which are invoked from *git fetch*, *git pull*, and *git clone*. If the client is authenticated, the *receive-pack* service is enabled, which serves *git send-pack* clients, which is invoked from *git push*.

SERVICES

These services can be enabled/disabled using the per-repository configuration file:

http.getanyfile

This serves Git clients older than version 1.6.6 that are unable to use the upload pack service. When enabled, clients are able to read any file within the repository, including objects that are no longer reachable from a branch but are still present. It is enabled by default, but a repository can disable it by setting this configuration value to *false*.

http.uploadpack

This serves *git fetch-pack* and *git ls-remote* clients. It is enabled by default, but a repository can disable it by setting this configuration value to *false*.

http.receivepack

This serves *git send-pack* clients, allowing push. It is disabled by default for anonymous users, and enabled by default for users authenticated by the web server. It can be disabled by setting this item to *false*, or enabled for all users, including anonymous users, by setting it to *true*.

URL TRANSLATION

To determine the location of the repository on disk, *git http-backend* concatenates the environment variables PATH_INFO, which is set automatically by the web server, and GIT_PROJECT_ROOT, which must be set manually in the web server configuration. If GIT_PROJECT_ROOT is not set, *git http-backend* reads PATH_TRANSLATED, which is also set automatically by the web server.

EXAMPLES

All of the following examples map *http://$hostname/git/foo/bar.git* to */var/www/git/foo/bar.git*.

Apache 2.x

Ensure mod_cgi, mod_alias, and mod_env are enabled, set GIT_PROJECT_ROOT (or DocumentRoot) appropriately, and create a ScriptAlias to the CGI:

```
SetEnv GIT_PROJECT_ROOT /var/www/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend/

# This is not strictly necessary using Apache and a modern version of
# git-http-backend, as the webserver will pass along the header in the
# environment as HTTP_GIT_PROTOCOL, and http-backend will copy that into
# GIT_PROTOCOL. But you may need this line (or something similar if you
# are using a different webserver), or if you want to support older Git
# versions that did not do that copying.
#
# Having the webserver set up GIT_PROTOCOL is perfectly fine even with
# modern versions (and will take precedence over HTTP_GIT_PROTOCOL,
# which means it can be used to override the client's request).
SetEnvIf Git-Protocol ".*" GIT_PROTOCOL=$0
```

To enable anonymous read access but authenticated write access, require authorization for both the initial ref advertisement (which we detect as a push via the service parameter in the query string), and the receive-pack invocation itself:

```
RewriteCond %{QUERY_STRING} service=git-receive-pack [OR]
RewriteCond %{REQUEST_URI} /git-receive-pack$
RewriteRule ^/git/ - [E=AUTHREQUIRED:yes]

<LocationMatch "^/git/">
        Order Deny,Allow
        Deny from env=AUTHREQUIRED

        AuthType Basic
        AuthName "Git Access"
        Require group committers
        Satisfy Any
        ...
</LocationMatch>
```

If you do not have *mod_rewrite* available to match against the query string, it is sufficient to just protect *git-receive-pack* itself, like:

```
<LocationMatch "^/git/.*/git-receive-pack$">
        AuthType Basic
        AuthName "Git Access"
        Require group committers
        ...
</LocationMatch>
```

In this mode, the server will not request authentication until the client actually starts the object negotiation phase of the push, rather than during the initial contact. For this reason, you must also enable the *http.receivepack* config option in any repositories that should accept a push. The default behavior, if *http.receivepack* is not set, is to reject any pushes by unauthenticated users; the initial request will therefore report *403 Forbidden* to the client, without even giving an opportunity for authentication.

To require authentication for both reads and writes, use a Location directive around the repository, or one of its parent directories:

```
<Location /git/private>
        AuthType Basic
        AuthName "Private Git Access"
        Require group committers
        ...
</Location>
```

To serve gitweb at the same url, use a ScriptAliasMatch to only those URLs that *git http-backend* can handle, and forward the rest to gitweb:

```
ScriptAliasMatch \
        "(?x)^/git/(.*/(HEAD | \
                        info/refs | \
                        objects/(info/[^/]+ | \
                                 [0-9a-f]{2}/[0-9a-f]{38} | \
                                 pack/pack-[0-9a-f]{40}\.(pack|idx)) | \
                        git-(upload|receive)-pack))$" \
        /usr/libexec/git-core/git-http-backend/$1

ScriptAlias /git/ /var/www/cgi-bin/gitweb.cgi/
```

To serve multiple repositories from different Section G.4.11, “gitnamespaces(7)” in a single repository:

```
SetEnvIf Request_URI "^/git/([^/]*)" GIT_NAMESPACE=$1
ScriptAliasMatch ^/git/[^/]*(.*) /usr/libexec/git-core/git-http-backend/
storage.git$1
```

Accelerated static Apache 2.x

Similar to the above, but Apache can be used to return static files that are stored on disk. On many systems this may be more efficient as Apache can ask the kernel to copy the file contents from the file system directly to the network:

```
SetEnv GIT_PROJECT_ROOT /var/www/git

AliasMatch ^/git/(.*/objects/[0-9a-f]{2}/[0-9a-f]{38})$          /var/
www/git/$1
AliasMatch ^/git/(.*/objects/pack/pack-[0-9a-f]{40}.(pack|idx))$ /var/
www/git/$1
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend/
```

This can be combined with the gitweb configuration:

```
SetEnv GIT_PROJECT_ROOT /var/www/git

AliasMatch ^/git/(.*/objects/[0-9a-f]{2}/[0-9a-f]{38})$          /var/
www/git/$1
AliasMatch ^/git/(.*/objects/pack/pack-[0-9a-f]{40}.(pack|idx))$ /var/
www/git/$1
ScriptAliasMatch \
        "(?x)^/git/(.*/(HEAD | \
                        info/refs | \
                        objects/info/[^/]+ | \
                        git-(upload|receive)-pack))$" \
        /usr/libexec/git-core/git-http-backend/$1
ScriptAlias /git/ /var/www/cgi-bin/gitweb.cgi/
```

Lighttpd

Ensure that *mod_cgi*, *mod_alias*, *mod_auth*, *mod_setenv* are loaded, then set *GIT_PROJECT_ROOT* appropriately and redirect all requests to the CGI:

```
alias.url += ( "/git" => "/usr/lib/git-core/git-http-backend" )
$HTTP["url"] =~ "^/git" {
        cgi.assign = ("" => "")
        setenv.add-environment = (
                "GIT_PROJECT_ROOT" => "/var/www/git",
                "GIT_HTTP_EXPORT_ALL" => ""
        )
}
```

To enable anonymous read access but authenticated write access:

```
$HTTP["querystring"] =~ "service=git-receive-pack" {
        include "git-auth.conf"
}
$HTTP["url"] =~ "^/git/.*/git-receive-pack$" {
        include "git-auth.conf"
}
```

where *git-auth.conf* looks something like:

```
auth.require = (
        "/" => (
                "method" => "basic",
                "realm" => "Git Access",
                "require" => "valid-user"
               )
```

```
)
# ...and set up auth.backend here
```

To require authentication for both reads and writes:

```
$HTTP["url"] =~ "^/git/private" {
        include "git-auth.conf"
}
```

ENVIRONMENT

git http-backend relies upon the *CGI* environment variables set by the invoking web server, including:

- PATH_INFO (if GIT_PROJECT_ROOT is set, otherwise PATH_TRANSLATED)
- REMOTE_USER
- REMOTE_ADDR
- CONTENT_TYPE
- QUERY_STRING
- REQUEST_METHOD

The *GIT_HTTP_EXPORT_ALL* environment variable may be passed to *git-http-backend* to bypass the check for the "git-daemon-export-ok" file in each repository before allowing export of that repository.

The *GIT_HTTP_MAX_REQUEST_BUFFER* environment variable (or the *http.maxRequestBuffer* config option) may be set to change the largest ref negotiation request that git will handle during a fetch; any fetch requiring a larger buffer will not succeed. This value should not normally need to be changed, but may be helpful if you are fetching from a repository with an extremely large number of refs. The value can be specified with a unit (e.g., *100M* for 100 megabytes). The default is 10 megabytes.

Clients may probe for optional protocol capabilities (like the v2 protocol) using the *Git-Protocol* HTTP header. In order to support these, the contents of that header must appear in the *GIT_PROTOCOL* environment variable. Most webservers will pass this header to the CGI via the *HTTP_GIT_PROTOCOL* variable, and *git-http-backend* will automatically copy that to *GIT_PROTOCOL*. However, some webservers may be more selective about which headers they'll pass, in which case they need to be configured explicitly (see the mention of *Git-Protocol* in the Apache config from the earlier EXAMPLES section).

The backend process sets GIT_COMMITTER_NAME to *$REMOTE_USER* and GIT_COMMITTER_EMAIL to *${REMOTE_USER}@http.${REMOTE_ADDR}*, ensuring that any reflogs created by *git-receive-pack* contain some identifying information of the remote user who performed the push.

All *CGI* environment variables are available to each of the hooks invoked by the *git-receive-pack*.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.66. git-http-fetch(1)

2024-07-02

Revision History

2024-07-02

NAME

git-http-fetch - Download from a remote Git repository via HTTP

SYNOPSIS

git http-fetch [-c] [-t] [-a] [-d] [-v] [-w <filename>] [--recover] [--stdin | --packfile=<hash> | <commit>] <URL>

DESCRIPTION

Downloads a remote Git repository via HTTP.

This command always gets all objects. Historically, there were three options *-a*, *-c* and *-t* for choosing which objects to download. They are now silently ignored.

OPTIONS

commit-id

Either the hash or the filename under [URL]/refs/ to pull.

-a, -c, -t

These options are ignored for historical reasons.

-v

Report what is downloaded.

-w <filename>

Writes the commit-id into the specified filename under $GIT_DIR/refs/<filename> on the local end after the transfer is complete.

--stdin

Instead of a commit id on the command line (which is not expected in this case), *git http-fetch* expects lines on stdin in the format

```
<commit-id>['\t'<filename-as-in--w>]
```

--packfile=<hash>

For internal use only. Instead of a commit id on the command line (which is not expected in this case), *git http-fetch* fetches the packfile directly at the given URL and uses index-pack to generate corresponding .idx and .keep files. The hash is used to determine the name of the temporary file and is arbitrary. The output of index-pack is printed to stdout. Requires --index-pack-args.

--index-pack-args=<args>

For internal use only. The command to run on the contents of the downloaded pack. Arguments are URL-encoded separated by spaces.

--recover

Verify that everything reachable from target is fetched. Used after an earlier fetch is interrupted.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.67. git-http-push(1)

2024-07-02
Revision History
2024-07-02

NAME

git-http-push - Push objects over HTTP/DAV to another repository

SYNOPSIS

git http-push [--all] [--dry-run] [--force] [--verbose] <URL> <ref> [<ref>…]

DESCRIPTION

Sends missing objects to the remote repository, and updates the remote branch.

NOTE: This command is temporarily disabled if your libcurl is older than 7.16, as the combination has been reported not to work and sometimes corrupts the repository.

OPTIONS

--all

Do not assume that the remote repository is complete in its current state, and verify all objects in the entire local ref's history exist in the remote repository.

--force

Usually, the command refuses to update a remote ref that is not an ancestor of the local ref used to overwrite it. This flag disables the check. What this means is that the remote repository can lose commits; use it with care.

--dry-run

Do everything except actually send the updates.

--verbose

Report the list of objects being walked locally and the list of objects successfully sent to the remote repository.

-d , -D

Remove <ref> from remote repository. The specified branch cannot be the remote HEAD. If -d is specified, the following other conditions must also be met:

- Remote HEAD must resolve to an object that exists locally
- Specified branch resolves to an object that exists locally
- Specified branch is an ancestor of the remote HEAD

<ref>…

The remote refs to update.

SPECIFYING THE REFS

A <*ref*> specification can be either a single pattern, or a pair of such patterns separated by a colon ":" (this means that a ref name cannot have a colon in it). A single pattern <*name*> is just a shorthand for <*name*>:<*name*>.

Each pattern pair <*src*>:<*dst*> consists of the source side (before the colon) and the destination side (after the colon). The ref to be pushed is determined by finding a match that matches the source side, and where it is pushed is determined by using the destination side.

- It is an error if <*src*> does not match exactly one of the local refs.
- If <*dst*> does not match any remote ref, either
  - it has to start with "refs/"; <dst> is used as the destination literally in this case.
  - <src> == <dst> and the ref that matched the <src> must not exist in the set of remote refs; the ref matched <src> locally is used as the name of the destination.

Without --force`, the <src> ref is stored at the remote only if <dst> does not exist, or <dst> is a proper subset (i.e. an ancestor) of <src>. This check, known as "fast-forward check", is performed to avoid accidentally overwriting the remote ref and losing other peoples commits from there.

With *--force*, the fast-forward check is disabled for all refs.

Optionally, a <ref> parameter can be prefixed with a plus + sign to disable the fast-forward check only on that ref.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.68. git-imap-send(1)

2024-07-02
Revision History
2024-07-02

NAME

git-imap-send - Send a collection of patches from stdin to an IMAP folder

SYNOPSIS

git imap-send [-v] [-q] [--[no-]curl]

DESCRIPTION

This command uploads a mailbox generated with *git format-patch* into an IMAP drafts folder. This allows patches to be sent as other email is when using mail clients that cannot read mailbox files directly. The command also works with any general mailbox in which emails have the fields "From", "Date", and "Subject" in that order.

Typical usage is something like:

git format-patch --signoff --stdout --attach origin | git imap-send

OPTIONS

-v , --verbose

Be verbose.

-q , --quiet

Be quiet.

--curl

Use libcurl to communicate with the IMAP server, unless tunneling into it. Ignored if Git was built without the USE_CURL_FOR_IMAP_SEND option set.

--no-curl

Talk to the IMAP server using git's own IMAP routines instead of using libcurl. Ignored if Git was built with the NO_OPENSSL option set.

CONFIGURATION

To use the tool, *imap.folder* and either *imap.tunnel* or *imap.host* must be set to appropriate values.

Everything above this line in this section isn't included from the Section G.3.29, “git-config(1)” documentation. The content that follows is the same as what's found there:

imap.folder

The folder to drop the mails into, which is typically the Drafts folder. For example: "INBOX.Drafts", "INBOX/Drafts" or "[Gmail]/Drafts". Required.

imap.tunnel

Command used to set up a tunnel to the IMAP server through which commands will be piped instead of using a direct network connection to the server. Required when imap.host is not set.

imap.host

A URL identifying the server. Use an *imap://* prefix for non-secure connections and an *imaps://* prefix for secure connections. Ignored when imap.tunnel is set, but required otherwise.

imap.user

The username to use when logging in to the server.

imap.pass

The password to use when logging in to the server.

imap.port

An integer port number to connect to on the server. Defaults to 143 for imap:// hosts and 993 for imaps:// hosts. Ignored when imap.tunnel is set.

imap.sslverify

A boolean to enable/disable verification of the server certificate used by the SSL/TLS connection. Default is *true*. Ignored when imap.tunnel is set.

imap.preformattedHTML

A boolean to enable/disable the use of html encoding when sending a patch. An html encoded patch will be bracketed with <pre> and have a content type of text/html. Ironically, enabling this option causes Thunderbird to send the patch as a plain/text, format=fixed email. Default is *false*.

imap.authMethod

Specify the authentication method for authenticating with the IMAP server. If Git was built with the NO_CURL option, or if your curl version is older than 7.34.0, or if you're running git-imap-send with the *--no-curl* option, the only supported method is *CRAM-MD5*. If this is not set then *git imap-send* uses the basic IMAP plaintext LOGIN command.

EXAMPLES

Using tunnel mode:

```
[imap]
    folder = "INBOX.Drafts"
    tunnel = "ssh -q -C user@example.com /usr/bin/imapd ./Maildir 2> /dev/
null"
```

Using direct mode:

```
[imap]
    folder = "INBOX.Drafts"
    host = imap://imap.example.com
    user = bob
    pass = p4ssw0rd
```

Using direct mode with SSL:

```
[imap]
    folder = "INBOX.Drafts"
    host = imaps://imap.example.com
    user = bob
    pass = p4ssw0rd
    port = 123
    ; sslVerify = false
```

> **Note**
>
> You may want to use *sslVerify=false* while troubleshooting, if you suspect that the reason you are having trouble connecting is because the certificate you use at the private server *example.com* you are trying to set up (or have set up) may not be verified correctly.

Using Gmail's IMAP interface:

```
[imap]
        folder = "[Gmail]/Drafts"
        host = imaps://imap.gmail.com
        user = user@gmail.com
        port = 993
```

> **Note**
>
> You might need to instead use: *folder = "[Google Mail]/Drafts"* if you get an error that the "Folder doesn't exist".

> **Note**
>
> If your Gmail account is set to another language than English, the name of the "Drafts" folder will be localized.

Once the commits are ready to be sent, run the following command:

```
$ git format-patch --cover-letter -M --stdout origin/master | git imap-send
```

Just make sure to disable line wrapping in the email client (Gmail's web interface will wrap lines no matter what, so you need to use a real IMAP client).

CAUTION

It is still your responsibility to make sure that the email message sent by your email program meets the standards of your project. Many projects do not like patches to be attached. Some mail agents will transform patches (e.g. wrap lines, send them as format=flowed) in ways that make them fail. You will get angry flames ridiculing you if you don't check this.

Thunderbird in particular is known to be problematic. Thunderbird users may wish to visit this web page for more information: https://kb.mozillazine.org/Plain_text_e-mail_-_Thunderbird#Completely_plain_email

SEE ALSO

Section G.3.54, “git-format-patch(1)”, Section G.3.124, “git-send-email(1)”, mbox(5)

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.69. git-index-pack(1)

2024-07-02

Revision History

2024-07-02

NAME

git-index-pack - Build pack index file for an existing packed archive

SYNOPSIS

git index-pack [-v] [-o <index-file>] [--[no-]rev-index] <pack-file>
git index-pack --stdin [--fix-thin] [--keep] [-v] [-o <index-file>]
[--[no-]rev-index] [<pack-file>]

DESCRIPTION

Reads a packed archive (.pack) from the specified file, builds a pack index file (.idx) for it, and optionally writes a reverse-index (.rev) for the specified pack. The packed archive, together with the pack index, can then be placed in the objects/pack/ directory of a Git repository.

OPTIONS

-v

Be verbose about what is going on, including progress status.

-o <index-file>

Write the generated pack index into the specified file. Without this option the name of pack index file is constructed from the name of packed archive file by replacing .pack with .idx (and the program fails if the name of packed archive does not end with .pack).

--[no-]rev-index

When this flag is provided, generate a reverse index (a *.rev* file) corresponding to the given pack. If *--verify* is given, ensure that the existing reverse index is correct. Takes precedence over *pack.writeReverseIndex*.

--stdin

When this flag is provided, the pack is read from stdin instead and a copy is then written to <pack-file>. If <pack-file> is not specified, the pack is written to objects/pack/ directory of the current Git repository with a default name determined from the pack content. If <pack-file> is not specified consider using --keep to prevent a race condition between this process and *git repack*.

--fix-thin

Fix a "thin" pack produced by *git pack-objects --thin* (see Section G.3.96, “git-pack-objects(1)” for details) by adding the excluded objects the deltified objects are based on to the pack. This option only makes sense in conjunction with --stdin.

--keep

Before moving the index into its final destination create an empty .keep file for the associated pack file. This option is usually necessary with --stdin to prevent a simultaneous *git repack* process from deleting the newly constructed pack and index before refs can be updated to use objects contained in the pack.

--keep=<msg>

Like --keep, create a .keep file before moving the index into its final destination. However, instead of creating an empty file place *<msg>* followed by an LF into the .keep file. The *<msg>* message can later be searched for within all .keep files to locate any which have outlived their usefulness.

--index-version=<version>[,<offset>]

This is intended to be used by the test suite only. It allows to force the version for the generated pack index, and to force 64-bit index entries on objects located above the given offset.

--strict[=<msg-id>=<severity>…]

Die, if the pack contains broken objects or links. An optional comma-separated list of *<msg-id>=<severity>* can be passed to change the severity of some possible issues, e.g., *--strict="missingEmail=ignore,badTagName=error"*. See the entry for the *fsck.<msg-id>* configuration options in Section G.3.56, “git-fsck(1)” for more information on the possible values of *<msg-id>* and *<severity>*.

--progress-title

For internal use only.

Set the title of the progress bar. The title is "Receiving objects" by default and "Indexing objects" when *--stdin* is specified.

--check-self-contained-and-connected

Die if the pack contains broken links. For internal use only.

--fsck-objects[=<msg-id>=<severity>…]

Die if the pack contains broken objects, but unlike *--strict*, don't choke on broken links. If the pack contains a tree pointing to a .gitmodules blob that does not exist, prints the hash of that blob (for the caller to check) after the hash that goes into the name of the pack/idx file (see "Notes").

An optional comma-separated list of *<msg-id>=<severity>* can be passed to change the severity of some possible issues, e.g., *--fsck-objects="missingEmail=ignore,badTagName=ignore"*. See the entry for the *fsck.<msg-id>* configuration options in Section G.3.56, “git-fsck(1)” for more information on the possible values of *<msg-id>* and *<severity>*.

--threads=<n>

Specifies the number of threads to spawn when resolving deltas. This requires that index-pack be compiled with pthreads otherwise this option is ignored with a warning. This is meant to reduce packing time on multiprocessor machines. The required amount of memory for the delta search window is however multiplied by the number of threads. Specifying 0 will cause Git to auto-detect the number of CPU's and use maximum 3 threads.

--max-input-size=<size>

Die, if the pack is larger than <size>.

--object-format=<hash-algorithm>

Specify the given object format (hash algorithm) for the pack. The valid values are *sha1* and (if enabled) *sha256*. The default is the algorithm for the current repository (set by *extensions.objectFormat*), or *sha1* if no value is set or outside a repository.

This option cannot be used with --stdin.

Note: At present, there is no interoperability between SHA-256 repositories and SHA-1 repositories.

Historically, we warned that SHA-256 repositories may later need backward incompatible changes when we introduce such interoperability features. Today, we only expect compatible changes. Furthermore, if such changes prove to be necessary, it can be expected that SHA-256 repositories created with today's Git will be usable by future versions of Git without data loss.

--promisor[=<message>]

Before committing the pack-index, create a .promisor file for this pack. Particularly helpful when writing a promisor pack with --fix-thin since the name of the pack is not final until the pack has been fully written. If a *<message>* is provided, then that content will be written to the .promisor file for future reference. See *partial clone* [https://www.kernel.org/pub/software/scm/git/docs/technical/partial-clone.html] for more information.

NOTES

Once the index has been created, the hash that goes into the name of the pack/idx file is printed to stdout. If --stdin was also used then this is prefixed by either "pack\t", or "keep\t" if a new .keep file was successfully created. This is useful to remove a .keep file used as a lock to prevent the race with *git repack* mentioned above.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.70. git-init-db(1)

2024-07-02
Revision History
2024-07-02

NAME

git-init-db - Creates an empty Git repository

SYNOPSIS

git init-db [-q | --quiet] [--bare] [--template=<template-directory>] [--separate-git-dir <git-dir>] [--shared[=<permissions>

DESCRIPTION

This is a synonym for Section G.3.71, “git-init(1)”. Please refer to the documentation of that command.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.71. git-init(1)

2024-07-02
Revision History
2024-07-02

NAME

git-init - Create an empty Git repository or reinitialize an existing one

SYNOPSIS

git init [*-q* | *--quiet*] [*--bare*] [`--template=`*<template-directory>*]
[*--separate-git-dir <git-dir>*] [`--object-format=`*<format>*]
[`--ref-format=`*<format>*]
[*-b <branch-name>* | `--initial-branch=`*<branch-name>*]
[`--shared`[=*<permissions>*]] [*<directory>*]

DESCRIPTION

This command creates an empty Git repository - basically a *.git* directory with subdirectories for *objects*, *refs/heads*, *refs/tags*, and template files. An initial branch without any commits will be created (see the *--initial-branch* option below for its name).

If the *$GIT_DIR* environment variable is set then it specifies a path to use instead of *./.git* for the base of the repository.

If the object storage directory is specified via the *$GIT_OBJECT_DIRECTORY* environment variable then the sha1 directories are created underneath; otherwise, the default *$GIT_DIR/objects* directory is used.

Running *git init* in an existing repository is safe. It will not overwrite things that are already there. The primary reason for rerunning *git init* is to pick up newly added templates (or to move the repository to another place if *--separate-git-dir* is given).

OPTIONS

-q , *--quiet*

Only print error and warning messages; all other output will be suppressed.

--bare

Create a bare repository. If *GIT_DIR* environment is not set, it is set to the current working directory.

`--object-format=`*<format>*

Specify the given object *<format>* (hash algorithm) for the repository. The valid values are *sha1* and (if enabled) *sha256*. *sha1* is the default.

Note: At present, there is no interoperability between SHA-256 repositories and SHA-1 repositories.

Historically, we warned that SHA-256 repositories may later need backward incompatible changes when we introduce such interoperability features. Today, we only expect compatible changes. Furthermore, if such changes prove to be necessary, it can be expected that SHA-256 repositories created with today's Git will be usable by future versions of Git without data loss.

`--ref-format=`*<format>*

Specify the given ref storage *<format>* for the repository. The valid values are:

- *files* for loose files with packed-refs. This is the default.
- *reftable* for the reftable format. This format is experimental and its internals are subject to change.

`--template=`*<template-directory>*

Specify the directory from which templates will be used. (See the "TEMPLATE DIRECTORY" section below.)

`--separate-git-dir=`*<git-dir>*

Instead of initializing the repository as a directory to either *$GIT_DIR* or *./.git/*, create a text file there containing the path to the actual repository. This file acts as a filesystem-agnostic Git symbolic link to the repository.

If this is a reinitialization, the repository will be moved to the specified path.

-b <branch-name> , `--initial-branch=`*<branch-name>*

Use *<branch-name>* for the initial branch in the newly created repository. If not specified, fall back to the default name (currently *master*, but this is subject to change in the future; the name can be customized via the *init.defaultBranch* configuration variable).

`--shared`[=(*false*|*true*|*umask*|*group*|*all*|*world*|*everybody*|*<perm>*)]

Specify that the Git repository is to be shared amongst several users. This allows users belonging to the same group to push into that repository. When specified, the config variable *core.sharedRepository* is set so that

files and directories under *$GIT_DIR* are created with the requested permissions. When not specified, Git will use permissions reported by *umask*(2).

The option can have the following values, defaulting to *group* if no value is given:

umask , *false*

Use permissions reported by *umask*(2). The default, when *--shared* is not specified.

group , *true*

Make the repository group-writable, (and *g+sx*, since the git group may not be the primary group of all users). This is used to loosen the permissions of an otherwise safe *umask*(2) value. Note that the umask still applies to the other permission bits (e.g. if umask is *0022*, using *group* will not remove read privileges from other (non-group) users). See *0xxx* for how to exactly specify the repository permissions.

all , *world* , *everybody*

Same as *group*, but make the repository readable by all users.

<perm>

<perm> is a 3-digit octal number prefixed with 0` and each file will have mode *<perm>*. *<perm>* will override users *umask*(2) value (and not only loosen permissions as *group* and *all* do). *0640* will create a repository which is group-readable, but not group-writable or accessible to others. *0660* will create a repo that is readable and writable to the current user and group, but inaccessible to others (directories and executable files get their *x* bit from the *r* bit for corresponding classes of users).

By default, the configuration flag *receive.denyNonFastForwards* is enabled in shared repositories, so that you cannot force a non fast-forwarding push into it.

If you provide a *<directory>*, the command is run inside it. If this directory does not exist, it will be created.

TEMPLATE DIRECTORY

Files and directories in the template directory whose name do not start with a dot will be copied to the *$GIT_DIR* after it is created.

The template directory will be one of the following (in order):

- the argument given with the *--template* option;
- the contents of the *$GIT_TEMPLATE_DIR* environment variable;
- the *init.templateDir* configuration variable; or
- the default template directory: */usr/share/git-core/templates*.

The default template directory includes some directory structure, suggested "exclude patterns" (see Section G.4.5, “gitignore(5)”), and sample hook files.

The sample hooks are all disabled by default. To enable one of the sample hooks rename it by removing its *.sample* suffix.

See Section G.4.7, “githooks(5)” for more general info on hook execution.

EXAMPLES

Start a new Git repository for an existing code base

```
$ cd /path/to/my/codebase
$ git init      ❶
```

```
$ git add .      ❷
$ git commit     ❸
```

❶ Create a */path/to/my/codebase/.git* directory.
❷ Add all existing files to the index.
❸ Record the pristine state as the first commit in the history.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

init.templateDir

Specify the directory from which templates will be copied.

init.defaultBranch

Allows overriding the default branch name e.g. when initializing a new repository.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.72. git-instaweb(1)

2024-07-02
Revision History
2024-07-02

NAME

git-instaweb - Instantly browse your working repository in gitweb

SYNOPSIS

git instaweb [--local] [--httpd=<httpd>] [--port=<port>]
[--browser=<browser>]
git instaweb [--start] [--stop] [--restart]

DESCRIPTION

A simple script to set up *gitweb* and a web server for browsing the local repository.

OPTIONS

-l , --local

Only bind the web server to the local IP (127.0.0.1).

-d , --httpd

The HTTP daemon command-line that will be executed. Command-line options may be specified here, and the configuration file will be added at the end of the command-line. Currently apache2, lighttpd, mongoose, plackup, python and webrick are supported. (Default: lighttpd)

-m , --module-path

The module path (only needed if httpd is Apache). (Default: /usr/lib/apache2/modules)

-p , --port

The port number to bind the httpd to. (Default: 1234)

-b , --browser

The web browser that should be used to view the gitweb page. This will be passed to the *git web--browse* helper script along with the URL of the gitweb instance. See Section G.3.157, “git-web--browse(1)” for more information about this. If the script fails, the URL will be printed to stdout.

start , --start

Start the httpd instance and exit. Regenerate configuration files as necessary for spawning a new instance.

stop , --stop

Stop the httpd instance and exit. This does not generate any of the configuration files for spawning a new instance, nor does it close the browser.

restart , --restart

Restart the httpd instance and exit. Regenerate configuration files as necessary for spawning a new instance.

CONFIGURATION

You may specify configuration in your .git/config

```
[instaweb]
        local = true
        httpd = apache2 -f
        port = 4321
        browser = konqueror
        modulePath = /usr/lib/apache2/modules
```

If the configuration variable *instaweb.browser* is not set, *web.browser* will be used instead if it is defined. See Section G.3.157, “git-web--browse(1)” for more information about this.

SEE ALSO

Section G.4.16, “gitweb(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.73. git-interpret-trailers(1)

2024-07-02

Revision History

2024-07-02

NAME

git-interpret-trailers - Add or parse structured information in commit messages

SYNOPSIS

git interpret-trailers [--in-place] [--trim-empty]
[(--trailer (<key>|<key-alias>)[(=|:)<value>])…]
[--parse] [<file>…]

DESCRIPTION

Add or parse *trailer* lines that look similar to RFC 822 e-mail headers, at the end of the otherwise free-form part of a commit message. For example, in the following commit message

```
subject

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Signed-off-by: Alice <alice@example.com>
Signed-off-by: Bob <bob@example.com>
```

the last two lines starting with "Signed-off-by" are trailers.

This command reads commit messages from either the <file> arguments or the standard input if no <file> is specified. If *--parse* is specified, the output consists of the parsed trailers coming from the input, without influencing them with any command line options or configuration variables.

Otherwise, this command applies *trailer.** configuration variables (which could potentially add new trailers, as well as reposition them), as well as any command line arguments that can override configuration variables (such as *--trailer=...* which could also add new trailers), to each input file. The result is emitted on the standard output.

This command can also operate on the output of Section G.3.54, “git-format-patch(1)”, which is more elaborate than a plain commit message. Namely, such output includes a commit message (as above), a "---" divider line, and a patch part. For these inputs, the divider and patch parts are not modified by this command and are emitted as is on the output, unless *--no-divider* is specified.

Some configuration variables control the way the *--trailer* arguments are applied to each input and the way any existing trailer in the input is changed. They also make it possible to automatically add some trailers.

By default, a *<key>=<value>* or *<key>:<value>* argument given using *--trailer* will be appended after the existing trailers only if the last trailer has a different (<key>, <value>) pair (or if there is no existing trailer). The <key> and <value> parts will be trimmed to remove starting and trailing whitespace, and the resulting trimmed <key> and <value> will appear in the output like this:

```
key: value
```

This means that the trimmed <key> and <value> will be separated by *': '* (one colon followed by one space).

For convenience, a <key-alias> can be configured to make using *--trailer* shorter to type on the command line. This can be configured using the *trailer.<key-alias>.key* configuration variable. The <keyAlias> must be a prefix of the full <key> string, although case sensitivity does not matter. For example, if you have

```
trailer.sign.key "Signed-off-by: "
```

in your configuration, you only need to specify *--trailer="sign: foo"* on the command line instead of *--trailer="Signed-off-by: foo"*.

By default the new trailer will appear at the end of all the existing trailers. If there is no existing trailer, the new trailer will appear at the end of the input. A blank line will be added before the new trailer if there isn't one already.

Existing trailers are extracted from the input by looking for a group of one or more lines that (i) is all trailers, or (ii) contains at least one Git-generated or user-configured trailer and consists of at least 25% trailers. The group must be preceded by one or more empty (or whitespace-only) lines. The group must either be at the end of the input or be the last non-whitespace lines before a line that starts with --- (followed by a space or the end of the line).

When reading trailers, there can be no whitespace before or inside the <key>, but any number of regular space and tab characters are allowed between the <key> and the separator. There can be whitespaces before, inside or after the <value>. The <value> may be split over multiple lines with each subsequent line starting with at least one whitespace, like the "folding" in RFC 822. Example:

```
key: This is a very long value, with spaces and
  newlines in it.
```

Note that trailers do not follow (nor are they intended to follow) many of the rules for RFC 822 headers. For example they do not follow the encoding rule.

OPTIONS

--in-place

Edit the files in place.

--trim-empty

If the <value> part of any trailer contains only whitespace, the whole trailer will be removed from the output. This applies to existing trailers as well as new trailers.

--trailer <key>[(=|:)<value>]

Specify a (<key>, <value>) pair that should be applied as a trailer to the inputs. See the description of this command.

--where <placement> , --no-where

Specify where all new trailers will be added. A setting provided with *--where* overrides the *trailer.where* and any applicable *trailer.<keyAlias>.where* configuration variables and applies to all *--trailer* options until the next occurrence of *--where* or *--no-where*. Upon encountering *--no-where*, clear the effect of any previous use of *--where*, such that the relevant configuration variables are no longer overridden. Possible placements are *after*, *before*, *end* or *start*.

--if-exists <action> , --no-if-exists

Specify what action will be performed when there is already at least one trailer with the same <key> in the input. A setting provided with *--if-exists* overrides the *trailer.ifExists* and any applicable *trailer.<keyAlias>.ifExists* configuration variables and applies to all *--trailer* options until the next occurrence of *--if-exists* or *--no-if-exists*. Upon encountering '--no-if-exists, clear the effect of any previous use of '--if-exists, such that the relevant configuration variables are no longer overridden. Possible actions are *addIfDifferent*, *addIfDifferentNeighbor*, *add*, *replace* and *doNothing*.

--if-missing <action> , --no-if-missing

Specify what action will be performed when there is no other trailer with the same <key> in the input. A setting provided with *--if-missing* overrides the *trailer.ifMissing* and any applicable *trailer.<keyAlias>.ifMissing* configuration variables and applies to all *--trailer* options until the next occurrence of *--if-missing* or *--no-if-missing*. Upon encountering '--no-if-missing, clear the effect of any previous use of '--if-missing, such that the relevant configuration variables are no longer overridden. Possible actions are *doNothing* or *add*.

--only-trailers

Output only the trailers, not any other parts of the input.

--only-input

Output only trailers that exist in the input; do not add any from the command-line or by applying *trailer.** configuration variables.

--unfold

If a trailer has a value that runs over multiple lines (aka "folded"), reformat the value into a single line.

--parse

A convenience alias for *--only-trailers --only-input --unfold*. This makes it easier to only see the trailers coming from the input without influencing them with any command line options or configuration variables, while also making the output machine-friendly with --unfold.

--no-divider

Do not treat --- as the end of the commit message. Use this when you know your input contains just the commit message itself (and not an email or the output of *git format-patch*).

CONFIGURATION VARIABLES

trailer.separators

This option tells which characters are recognized as trailer separators. By default only *:* is recognized as a trailer separator, except that = is always accepted on the command line for compatibility with other git commands.

The first character given by this option will be the default character used when another separator is not specified in the config for this trailer.

For example, if the value for this option is "%=$", then only lines using the format *<key><sep><value>* with <sep> containing *%*, = or *$* and then spaces will be considered trailers. And *%* will be the default separator used, so by default trailers will appear like: *<key>% <value>* (one percent sign and one space will appear between the key and the value).

trailer.where

This option tells where a new trailer will be added.

This can be *end*, which is the default, *start*, *after* or *before*.

If it is *end*, then each new trailer will appear at the end of the existing trailers.

If it is *start*, then each new trailer will appear at the start, instead of the end, of the existing trailers.

If it is *after*, then each new trailer will appear just after the last trailer with the same <key>.

If it is *before*, then each new trailer will appear just before the first trailer with the same <key>.

trailer.ifexists

This option makes it possible to choose what action will be performed when there is already at least one trailer with the same <key> in the input.

The valid values for this option are: *addIfDifferentNeighbor* (this is the default), *addIfDifferent*, *add*, *replace* or *doNothing*.

With *addIfDifferentNeighbor*, a new trailer will be added only if no trailer with the same (<key>, <value>) pair is above or below the line where the new trailer will be added.

With *addIfDifferent*, a new trailer will be added only if no trailer with the same (<key>, <value>) pair is already in the input.

With *add*, a new trailer will be added, even if some trailers with the same (<key>, <value>) pair are already in the input.

With *replace*, an existing trailer with the same <key> will be deleted and the new trailer will be added. The deleted trailer will be the closest one (with the same <key>) to the place where the new one will be added.

With *doNothing*, nothing will be done; that is no new trailer will be added if there is already one with the same <key> in the input.

trailer.ifmissing

This option makes it possible to choose what action will be performed when there is not yet any trailer with the same <key> in the input.

The valid values for this option are: *add* (this is the default) and *doNothing*.

With *add*, a new trailer will be added.

With *doNothing*, nothing will be done.

trailer.<keyAlias>.key

Defines a <keyAlias> for the <key>. The <keyAlias> must be a prefix (case does not matter) of the <key>. For example, in *git config trailer.ack.key "Acked-by"* the "Acked-by" is the <key> and the "ack" is the <keyAlias>. This configuration allows the shorter *--trailer "ack:..."* invocation on the command line using the "ack" <keyAlias> instead of the longer *--trailer "Acked-by:..."*.

At the end of the <key>, a separator can appear and then some space characters. By default the only valid separator is *:*, but this can be changed using the *trailer.separators* config variable.

If there is a separator in the key, then it overrides the default separator when adding the trailer.

trailer.<keyAlias>.where

This option takes the same values as the *trailer.where* configuration variable and it overrides what is specified by that option for trailers with the specified <keyAlias>.

trailer.<keyAlias>.ifexists

This option takes the same values as the *trailer.ifexists* configuration variable and it overrides what is specified by that option for trailers with the specified <keyAlias>.

trailer.<keyAlias>.ifmissing

This option takes the same values as the *trailer.ifmissing* configuration variable and it overrides what is specified by that option for trailers with the specified <keyAlias>.

trailer.<keyAlias>.command

Deprecated in favor of *trailer.<keyAlias>.cmd*. This option behaves in the same way as *trailer.<keyAlias>.cmd*, except that it doesn't pass anything as argument to the specified command. Instead the first occurrence of substring $ARG is replaced by the <value> that would be passed as argument.

Note that $ARG in the user's command is only replaced once and that the original way of replacing $ARG is not safe.

When both *trailer.<keyAlias>.cmd* and *trailer.<keyAlias>.command* are given for the same <keyAlias>, *trailer.<keyAlias>.cmd* is used and *trailer.<keyAlias>.command* is ignored.

trailer.<keyAlias>.cmd

This option can be used to specify a shell command that will be called once to automatically add a trailer with the specified <keyAlias>, and then called each time a *--trailer <keyAlias>=<value>* argument is specified to modify the <value> of the trailer that this option would produce.

When the specified command is first called to add a trailer with the specified <keyAlias>, the behavior is as if a special *--trailer <keyAlias>=<value>* argument was added at the beginning of the "git interpret-trailers" command, where <value> is taken to be the standard output of the command with any leading and trailing whitespace trimmed off.

If some *--trailer <keyAlias>=<value>* arguments are also passed on the command line, the command is called again once for each of these arguments with the same <keyAlias>. And the <value> part of these arguments, if any, will be passed to the command as its first argument. This way the command can produce a <value> computed from the <value> passed in the *--trailer <keyAlias>=<value>* argument.

EXAMPLES

- Configure a *sign* trailer with a *Signed-off-by* key, and then add two of these trailers to a commit message file:

```
$ git config trailer.sign.key "Signed-off-by"
```

```
$ cat msg.txt
subject

body text
$ git interpret-trailers --trailer 'sign: Alice <alice@example.com>' --
trailer 'sign: Bob <bob@example.com>' <msg.txt
subject

body text

Signed-off-by: Alice <alice@example.com>
Signed-off-by: Bob <bob@example.com>
```

- Use the *--in-place* option to edit a commit message file in place:

```
$ cat msg.txt
subject

body text

Signed-off-by: Bob <bob@example.com>
$ git interpret-trailers --trailer 'Acked-by: Alice <alice@example.com>'
 --in-place msg.txt
$ cat msg.txt
subject

body text

Signed-off-by: Bob <bob@example.com>
Acked-by: Alice <alice@example.com>
```

- Extract the last commit as a patch, and add a *Cc* and a *Reviewed-by* trailer to it:

```
$ git format-patch -1
0001-foo.patch
$ git interpret-trailers --trailer 'Cc: Alice <alice@example.com>'
 --trailer 'Reviewed-by: Bob <bob@example.com>' 0001-foo.patch >0001-
bar.patch
```

- Configure a *sign* trailer with a command to automatically add a 'Signed-off-by: ' with the author information only if there is no 'Signed-off-by: ' already, and show how it works:

```
$ cat msg1.txt
subject

body text
$ git config trailer.sign.key "Signed-off-by: "
$ git config trailer.sign.ifmissing add
$ git config trailer.sign.ifexists doNothing
$ git config trailer.sign.cmd 'echo "$(git config user.name) <$(git
 config user.email)>"'
$ git interpret-trailers --trailer sign <msg1.txt
subject

body text

Signed-off-by: Bob <bob@example.com>
$ cat msg2.txt
subject
```

```
body text

Signed-off-by: Alice <alice@example.com>
$ git interpret-trailers --trailer sign <msg2.txt
subject

body text

Signed-off-by: Alice <alice@example.com>
```

- Configure a *fix* trailer with a key that contains a # and no space after this character, and show how it works:

```
$ git config trailer.separators ":#"
$ git config trailer.fix.key "Fix #"
$ echo "subject" | git interpret-trailers --trailer fix=42
subject

Fix #42
```

- Configure a *help* trailer with a cmd use a script *glog-find-author* which search specified author identity from git log in git repository and show how it works:

```
$ cat ~/bin/glog-find-author
#!/bin/sh
test -n "$1" && git log --author="$1" --pretty="%an <%ae>" -1 || true
$ cat msg.txt
subject

body text
$ git config trailer.help.key "Helped-by: "
$ git config trailer.help.ifExists "addIfDifferentNeighbor"
$ git config trailer.help.cmd "~/bin/glog-find-author"
$ git interpret-trailers --trailer="help:Junio" --trailer="help:Couder"
 <msg.txt
subject

body text

Helped-by: Junio C Hamano <gitster@pobox.com>
Helped-by: Christian Couder <christian.couder@gmail.com>
```

- Configure a *ref* trailer with a cmd use a script *glog-grep* to grep last relevant commit from git log in the git repository and show how it works:

```
$ cat ~/bin/glog-grep
#!/bin/sh
test -n "$1" && git log --grep "$1" --pretty=reference -1 || true
$ cat msg.txt
subject

body text
$ git config trailer.ref.key "Reference-to: "
$ git config trailer.ref.ifExists "replace"
$ git config trailer.ref.cmd "~/bin/glog-grep"
$ git interpret-trailers --trailer="ref:Add copyright notices." <msg.txt
subject

body text
```

```
Reference-to: 8bc9a0c769 (Add copyright notices., 2005-04-07)
```

- Configure a *see* trailer with a command to show the subject of a commit that is related, and show how it works:

```
$ cat msg.txt
subject

body text

see: HEAD~2
$ cat ~/bin/glog-ref
#!/bin/sh
git log -1 --oneline --format="%h (%s)" --abbrev-commit --abbrev=14
$ git config trailer.see.key "See-also: "
$ git config trailer.see.ifExists "replace"
$ git config trailer.see.ifMissing "doNothing"
$ git config trailer.see.cmd "glog-ref"
$ git interpret-trailers --trailer=see <msg.txt
subject

body text

See-also: fe3187489d69c4 (subject of related commit)
```

- Configure a commit template with some trailers with empty values (using sed to show and keep the trailing spaces at the end of the trailers), then configure a commit-msg hook that uses *git interpret-trailers* to remove trailers with empty values and to add a *git-version* trailer:

```
$ cat temp.txt
***subject***

***message***

Fixes: Z
Cc: Z
Reviewed-by: Z
Signed-off-by: Z
$ sed -e 's/ Z$/ /' temp.txt > commit_template.txt
$ git config commit.template commit_template.txt
$ cat .git/hooks/commit-msg
#!/bin/sh
git interpret-trailers --trim-empty --trailer "git-version: \$(git
 describe)" "\$1" > "\$1.new"
mv "\$1.new" "\$1"
$ chmod +x .git/hooks/commit-msg
```

SEE ALSO

Section G.3.28, “git-commit(1)”, Section G.3.54, “git-format-patch(1)”, Section G.3.29, “git-config(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.74. git-log(1)

2024-07-02

Revision History

2024-07-02

NAME

git-log - Show commit logs

SYNOPSIS

git log [<options>] [<revision-range>] [[--] <path>…]

DESCRIPTION

Shows the commit logs.

List commits that are reachable by following the *parent* links from the given commit(s), but exclude commits that are reachable from the one(s) given with a ^ in front of them. The output is given in reverse chronological order by default.

You can think of this as a set operation. Commits reachable from any of the commits given on the command line form a set, and then commits reachable from any of the ones given with ^ in front are subtracted from that set. The remaining commits are what comes out in the command's output. Various other options and paths parameters can be used to further limit the result.

Thus, the following command:

```
$ git log foo bar ^baz
```

means "list all the commits which are reachable from *foo* or *bar*, but not from *baz*".

A special notation "*<commit1>*..*<commit2>*" can be used as a short-hand for "^*<commit1>* *<commit2>*". For example, either of the following may be used interchangeably:

```
$ git log origin..HEAD
$ git log HEAD ^origin
```

Another special notation is "*<commit1>*…*<commit2>*" which is useful for merges. The resulting set of commits is the symmetric difference between the two operands. The following two commands are equivalent:

```
$ git log A B --not $(git merge-base --all A B)
$ git log A...B
```

The command takes options applicable to the Section G.3.120, “git-rev-list(1)” command to control what is shown and how, and options applicable to the Section G.3.44, “git-diff(1)” command to control how the changes each commit introduces are shown.

OPTIONS

--follow

Continue listing the history of a file beyond renames (works only for a single file).

--no-decorate , --decorate[=short|full|auto|no]

Print out the ref names of any commits that are shown. If *short* is specified, the ref name prefixes *refs/heads/*, *refs/tags/* and *refs/remotes/* will not be printed. If *full* is specified, the full ref name (including prefix) will be printed. If *auto* is specified, then if the output is going to a terminal, the ref names are shown as if *short* were given, otherwise no ref names are shown. The option *--decorate* is short-hand for *--decorate=short*. Default to configuration value of *log.decorate* if configured, otherwise, *auto*.

--decorate-refs=<pattern> , --decorate-refs-exclude=<pattern>

For each candidate reference, do not use it for decoration if it matches any patterns given to *--decorate-refs-exclude* or if it doesn't match any of the patterns given to *--decorate-refs*. The *log.excludeDecoration* config option allows excluding refs from the decorations, but an explicit *--decorate-refs* pattern will override a match in *log.excludeDecoration*.

If none of these options or config settings are given, then references are used as decoration if they match *HEAD*, *refs/heads/*, *refs/remotes/*, *refs/stash/*, or *refs/tags/*.

--clear-decorations

When specified, this option clears all previous *--decorate-refs* or *--decorate-refs-exclude* options and relaxes the default decoration filter to include all references. This option is assumed if the config value *log.initialDecorationSet* is set to *all*.

--source

Print out the ref name given on the command line by which each commit was reached.

--[no-]mailmap , --[no-]use-mailmap

Use mailmap file to map author and committer names and email addresses to canonical real names and email addresses. See Section G.3.130, “git-shortlog(1)”.

--full-diff

Without this flag, *git log -p <path>...* shows commits that touch the specified paths, and diffs about the same specified paths. With this, the full diff is shown for commits that touch the specified paths; this means that "<path>…" limits only commits, and doesn't limit diff for those commits.

Note that this affects all diff-based output types, e.g. those produced by *--stat*, etc.

--log-size

Include a line log size <number> in the output for each commit, where <number> is the length of that commit's message in bytes. Intended to speed up tools that read log messages from *git log* output by allowing them to allocate space in advance.

-L<start>,<end>:<file> , -L:<funcname>:<file>

Trace the evolution of the line range given by *<start>,<end>*, or by the function name regex *<funcname>*, within the *<file>*. You may not give any pathspec limiters. This is currently limited to a walk starting from a single revision, i.e., you may only give zero or one positive revision arguments, and *<start>* and *<end>* (or *<funcname>*) must exist in the starting revision. You can specify this option more than once. Implies *--patch*. Patch output can be suppressed using *--no-patch*, but other diff formats (namely *--raw*, *--numstat*, *--shortstat*, *--dirstat*, *--summary*, *--name-only*, *--name-status*, *--check*) are not currently implemented.

<start> and *<end>* can take one of these forms:

- number

  If *<start>* or *<end>* is a number, it specifies an absolute line number (lines count from 1).

- */regex/*

  This form will use the first line matching the given POSIX regex. If *<start>* is a regex, it will search from the end of the previous *-L* range, if any, otherwise from the start of file. If *<start>* is *^/regex/*, it will search from the start of file. If *<end>* is a regex, it will search starting at the line given by *<start>*.

- +offset or -offset

  This is only valid for *<end>* and will specify a number of lines before or after the line given by *<start>*.

If *:<funcname>* is given in place of *<start>* and *<end>*, it is a regular expression that denotes the range from the first funcname line that matches *<funcname>*, up to the next funcname line. *:<funcname>* searches from the end of the previous *-L* range, if any, otherwise from the start of file. *^:<funcname>* searches from the start of file. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

<revision-range>

Show only commits in the specified revision range. When no <revision-range> is specified, it defaults to *HEAD* (i.e. the whole history leading to the current commit). *origin..HEAD* specifies all the commits reachable from the current commit (i.e. *HEAD*), but not from *origin*. For a complete list of ways to spell <revision-range>, see the *Specifying Ranges* section of Section G.4.14, “gitrevisions(7)”.

[--] <path>…

Show only commits that are enough to explain how the files that match the specified paths came to be. See *History Simplification* below for details and other simplification modes.

Paths may need to be prefixed with -- to separate them from options or the revision range, when confusion arises.

1. Commit Limiting

Besides specifying a range of commits that should be listed using the special notations explained in the description, additional commit limiting may be applied.

Using more options generally further limits the output (e.g. *--since=<date1>* limits to commits newer than *<date1>*, and using it with *--grep=<pattern>* further limits to commits whose log message has a line that matches *<pattern>*), unless otherwise noted.

Note that these are applied before commit ordering and formatting options, such as *--reverse*.

-<number> , -n <number> , --max-count=<number>

Limit the number of commits to output.

--skip=<number>

Skip *number* commits before starting to show the commit output.

--since=<date> , --after=<date>

Show commits more recent than a specific date.

--since-as-filter=<date>

Show all commits more recent than a specific date. This visits all commits in the range, rather than stopping at the first commit which is older than a specific date.

--until=<date> , --before=<date>

Show commits older than a specific date.

--author=<pattern> , --committer=<pattern>

Limit the commits output to ones with author/committer header lines that match the specified pattern (regular expression). With more than one *--author=<pattern>*, commits whose author matches any of the given patterns are chosen (similarly for multiple *--committer=<pattern>*).

--grep-reflog=<pattern>

Limit the commits output to ones with reflog entries that match the specified pattern (regular expression). With more than one *--grep-reflog*, commits whose reflog message matches any of the given patterns are chosen. It is an error to use this option unless *--walk-reflogs* is in use.

--grep=<pattern>

Limit the commits output to ones with a log message that matches the specified pattern (regular expression). With more than one *--grep=<pattern>*, commits whose message matches any of the given patterns are chosen (but see *--all-match*).

When *--notes* is in effect, the message from the notes is matched as if it were part of the log message.

--all-match

Limit the commits output to ones that match all given *--grep*, instead of ones that match at least one.

--invert-grep

Limit the commits output to ones with a log message that do not match the pattern specified with *--grep=<pattern>*.

-i , --regexp-ignore-case

Match the regular expression limiting patterns without regard to letter case.

--basic-regexp

Consider the limiting patterns to be basic regular expressions; this is the default.

-E , --extended-regexp

Consider the limiting patterns to be extended regular expressions instead of the default basic regular expressions.

-F , --fixed-strings

Consider the limiting patterns to be fixed strings (don't interpret pattern as a regular expression).

-P , --perl-regexp

Consider the limiting patterns to be Perl-compatible regular expressions.

Support for these types of regular expressions is an optional compile-time dependency. If Git wasn't compiled with support for them providing this option will cause it to die.

--remove-empty

Stop when a given path disappears from the tree.

--merges

Print only merge commits. This is exactly the same as *--min-parents=2*.

--no-merges

Do not print commits with more than one parent. This is exactly the same as *--max-parents=1*.

--min-parents=<number> , --max-parents=<number> , --no-min-parents , --no-max-parents

Show only commits which have at least (or at most) that many parent commits. In particular, *--max-parents=1* is the same as *--no-merges*, *--min-parents=2* is the same as *--merges*. *--max-parents=0* gives all root commits and *--min-parents=3* all octopus merges.

--no-min-parents and *--no-max-parents* reset these limits (to no limit) again. Equivalent forms are *--min-parents=0* (any commit has 0 or more parents) and *--max-parents=-1* (negative numbers denote no upper limit).

--first-parent

When finding commits to include, follow only the first parent commit upon seeing a merge commit. This option can give a better overview when viewing the evolution of a particular topic branch, because merges into a topic branch tend to be only about adjusting to updated upstream from time to time, and this option allows you to ignore the individual commits brought in to your history by such a merge.

This option also changes default diff format for merge commits to *first-parent*, see *--diff-merges=first-parent* for details.

--exclude-first-parent-only

When finding commits to exclude (with a ^), follow only the first parent commit upon seeing a merge commit. This can be used to find the set of changes in a topic branch from the point where it diverged from the remote branch, given that arbitrary merges can be valid topic branch changes.

--not

Reverses the meaning of the ^ prefix (or lack thereof) for all following revision specifiers, up to the next *--not*. When used on the command line before --stdin, the revisions passed through stdin will not be affected by it. Conversely, when passed via standard input, the revisions passed on the command line will not be affected by it.

--all

Pretend as if all the refs in *refs/*, along with *HEAD*, are listed on the command line as *<commit>*.

--branches[=<pattern>]

Pretend as if all the refs in *refs/heads* are listed on the command line as *<commit>*. If *<pattern>* is given, limit branches to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--tags[=<pattern>]

Pretend as if all the refs in *refs/tags* are listed on the command line as *<commit>*. If *<pattern>* is given, limit tags to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--remotes[=<pattern>]

Pretend as if all the refs in *refs/remotes* are listed on the command line as *<commit>*. If *<pattern>* is given, limit remote-tracking branches to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--glob=<glob-pattern>

Pretend as if all the refs matching shell glob *<glob-pattern>* are listed on the command line as *<commit>*. Leading *refs/*, is automatically prepended if missing. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--exclude=<glob-pattern>

Do not include refs matching *<glob-pattern>* that the next *--all*, *--branches*, *--tags*, *--remotes*, or *--glob* would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next *--all*, *--branches*, *--tags*, *--remotes*, or *--glob* option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with *refs/heads*, *refs/tags*, or *refs/remotes* when applied to *--branches*, *--tags*, or *--remotes*, respectively, and they must begin with *refs/* when applied to *--glob* or *--all*. If a trailing */** is intended, it must be given explicitly.

--exclude-hidden=[fetch|receive|uploadpack]

Do not include refs that would be hidden by *git-fetch*, *git-receive-pack* or *git-upload-pack* by consulting the appropriate *fetch.hideRefs*, *receive.hideRefs* or *uploadpack.hideRefs* configuration along with *transfer.hideRefs* (see Section G.3.29, “git-config(1)”). This option affects the next pseudo-ref option *--all* or *--glob* and is cleared after processing them.

--reflog

Pretend as if all objects mentioned by reflogs are listed on the command line as *<commit>*.

--alternate-refs

Pretend as if all objects mentioned as ref tips of alternate repositories were listed on the command line. An alternate repository is any repository whose object directory is specified in *objects/info/alternates*. The set of included objects may be modified by *core.alternateRefsCommand*, etc. See Section G.3.29, "git-config(1)".

--single-worktree

By default, all working trees will be examined by the following options when there are more than one (see Section G.3.159, "git-worktree(1)"): *--all*, *--reflog* and *--indexed-objects*. This option forces them to examine the current working tree only.

--ignore-missing

Upon seeing an invalid object name in the input, pretend as if the bad input was not given.

--bisect

Pretend as if the bad bisection ref *refs/bisect/bad* was listed and as if it was followed by *--not* and the good bisection refs *refs/bisect/good-** on the command line.

--stdin

In addition to getting arguments from the command line, read them from standard input as well. This accepts commits and pseudo-options like *--all* and *--glob=*. When a -- separator is seen, the following input is treated as paths and used to limit the result. Flags like *--not* which are read via standard input are only respected for arguments passed in the same way and will not influence any subsequent command line arguments.

--cherry-mark

Like *--cherry-pick* (see below) but mark equivalent commits with = rather than omitting them, and inequivalent ones with +.

--cherry-pick

Omit any commit that introduces the same change as another commit on the other side when the set of commits are limited with symmetric difference.

For example, if you have two branches, *A* and *B*, a usual way to list all commits on only one side of them is with *--left-right* (see the example below in the description of the *--left-right* option). However, it shows the commits that were cherry-picked from the other branch (for example, 3rd on b may be cherry-picked from branch A). With this option, such pairs of commits are excluded from the output.

--left-only , --right-only

List only commits on the respective side of a symmetric difference, i.e. only those which would be marked < resp. > by *--left-right*.

For example, *--cherry-pick --right-only A...B* omits those commits from *B* which are in *A* or are patch-equivalent to a commit in *A*. In other words, this lists the + commits from *git cherry A B*. More precisely, *--cherry-pick --right-only --no-merges* gives the exact list.

--cherry

A synonym for *--right-only --cherry-mark --no-merges*; useful to limit the output to the commits on our side and mark those that have been applied to the other side of a forked history with *git log --cherry upstream...mybranch*, similar to *git cherry upstream mybranch*.

-g , --walk-reflogs

Instead of walking the commit ancestry chain, walk reflog entries from the most recent one to older ones. When this option is used you cannot specify commits to exclude (that is, *^commit*, *commit1..commit2*, and *commit1...commit2* notations cannot be used).

With *--pretty* format other than *oneline* and *reference* (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. The reflog designator in the output may be shown as *ref@{<Nth>}* (where *<Nth>* is the reverse-chronological index in the reflog) or as *ref@{<timestamp>}* (with the *<timestamp>* for that entry), depending on a few rules:

1. If the starting point is specified as *ref@{<Nth>}*, show the index format.
2. If the starting point was specified as *ref@{now}*, show the timestamp format.
3. If neither was used, but *--date* was given on the command line, show the timestamp in the format requested by *--date*.
4. Otherwise, show the index format.

Under *--pretty=oneline*, the commit message is prefixed with this information on the same line. This option cannot be combined with *--reverse*. See also Section G.3.109, “git-reflog(1)”.

Under *--pretty=reference*, this information will not be shown at all.

--merge

Show commits touching conflicted paths in the range *HEAD...<other>*, where *<other>* is the first existing pseudoref in *MERGE_HEAD*, *CHERRY_PICK_HEAD*, *REVERT_HEAD* or *REBASE_HEAD*. Only works when the index has unmerged entries. This option can be used to show relevant commits when resolving conflicts from a 3-way merge.

--boundary

Output excluded boundary commits. Boundary commits are prefixed with -.

2. History Simplification

Sometimes you are only interested in parts of the history, for example the commits modifying a particular <path>. But there are two parts of *History Simplification*, one part is selecting the commits and the other is how to do it, as there are various strategies to simplify the history.

The following options select the commits to be shown:

<paths>

Commits modifying the given <paths> are selected.

--simplify-by-decoration

Commits that are referred by some branch or tag are selected.

Note that extra commits can be shown to give a meaningful history.

The following options affect the way the simplification is performed:

Default mode

Simplifies the history to the simplest history explaining the final state of the tree. Simplest because it prunes some side branches if the end result is the same (i.e. merging branches with the same content)

--show-pulls

Include all commits from the default mode, but also any merge commits that are not TREESAME to the first parent but are TREESAME to a later parent. This mode is helpful for showing the merge commits that "first introduced" a change to a branch.

--full-history

Same as the default mode, but does not prune some history.

--dense

Only the selected commits are shown, plus some to have a meaningful history.

--sparse

All commits in the simplified history are shown.

--simplify-merges

Additional option to *--full-history* to remove some needless merges from the resulting history, as there are no selected commits contributing to this merge.

--ancestry-path[=<commit>]

When given a range of commits to display (e.g. *commit1..commit2* or *commit2 ^commit1*), only display commits in that range that are ancestors of <commit>, descendants of <commit>, or <commit> itself. If no commit is specified, use *commit1* (the excluded part of the range) as <commit>. Can be passed multiple times; if so, a commit is included if it is any of the commits given or if it is an ancestor or descendant of one of them.

A more detailed explanation follows.

Suppose you specified *foo* as the <paths>. We shall call commits that modify *foo* !TREESAME, and the rest TREESAME. (In a diff filtered for *foo*, they look different and equal, respectively.)

In the following, we will always refer to the same example history to illustrate the differences between simplification settings. We assume that you are filtering for a file *foo* in this commit graph:

```
  .-A---M---N---O---P---Q
 /     /   /   /   /   /
I     B   C   D   E   Y
 \   /   /   /   /   /
  `-------------'   X
```

The horizontal line of history A---Q is taken to be the first parent of each merge. The commits are:

- *I* is the initial commit, in which *foo* exists with contents asdf, and a file *quux* exists with contents quux. Initial commits are compared to an empty tree, so *I* is !TREESAME.
- In *A*, *foo* contains just foo.
- *B* contains the same change as *A*. Its merge *M* is trivial and hence TREESAME to all parents.
- *C* does not change *foo*, but its merge *N* changes it to foobar, so it is not TREESAME to any parent.
- *D* sets *foo* to baz. Its merge *O* combines the strings from *N* and *D* to foobarbaz; i.e., it is not TREESAME to any parent.
- *E* changes *quux* to xyzzy, and its merge *P* combines the strings to quux xyzzy. *P* is TREESAME to *O*, but not to *E*.
- *X* is an independent root commit that added a new file *side*, and *Y* modified it. *Y* is TREESAME to *X*. Its merge *Q* added *side* to *P*, and *Q* is TREESAME to *P*, but not to *Y*.

rev-list walks backwards through history, including or excluding commits based on whether *--full-history* and/or parent rewriting (via *--parents* or *--children*) are used. The following settings are available.

Default mode

Commits are included if they are not TREESAME to any parent (though this can be changed, see *--sparse* below). If the commit was a merge, and it was TREESAME to one parent, follow only that parent. (Even if there are several TREESAME parents, follow only one of them.) Otherwise, follow all parents.

This results in:

```
  .-A---N---O
 /     /   /
I---------D
```

Note how the rule to only follow the TREESAME parent, if one is available, removed *B* from consideration entirely. *C* was considered via *N*, but is TREESAME. Root commits are compared to an empty tree, so *I* is !TREESAME.

Parent/child relations are only visible with *--parents*, but that does not affect the commits selected in default mode, so we have shown the parent lines.

--full-history without parent rewriting

This mode differs from the default in one point: always follow all parents of a merge, even if it is TREESAME to one of them. Even if more than one side of the merge has commits that are included, this does not imply that the merge itself is! In the example, we get

```
I  A  B  N  D  O  P  Q
```

M was excluded because it is TREESAME to both parents. *E*, *C* and *B* were all walked, but only *B* was !TREESAME, so the others do not appear.

Note that without parent rewriting, it is not really possible to talk about the parent/child relationships between the commits, so we show them disconnected.

--full-history with parent rewriting

Ordinary commits are only included if they are !TREESAME (though this can be changed, see *--sparse* below).

Merges are always included. However, their parent list is rewritten: Along each parent, prune away commits that are not included themselves. This results in

```
  .-A---M---N---O---P---Q
 /     /   /   /   /
I     B   /   D   /
 \   /   /   /   /
  `-------------'
```

Compare to *--full-history* without rewriting above. Note that *E* was pruned away because it is TREESAME, but the parent list of P was rewritten to contain *E*'s parent *I*. The same happened for *C* and *N*, and *X*, *Y* and *Q*.

In addition to the above settings, you can change whether TREESAME affects inclusion:

--dense

Commits that are walked are included if they are not TREESAME to any parent.

--sparse

All commits that are walked are included.

Note that without *--full-history*, this still simplifies merges: if one of the parents is TREESAME, we follow only that one, so the other sides of the merge are never walked.

--simplify-merges

First, build a history graph in the same way that *--full-history* with parent rewriting does (see above).

Then simplify each commit *C* to its replacement *C'* in the final history according to the following rules:

- Set *C'* to *C*.
- Replace each parent *P* of *C'* with its simplification *P'*. In the process, drop parents that are ancestors of other parents or that are root commits TREESAME to an empty tree, and remove duplicates, but take care to never drop all parents that we are TREESAME to.
- If after this parent rewriting, *C'* is a root or merge commit (has zero or >1 parents), a boundary commit, or !TREESAME, it remains. Otherwise, it is replaced with its only parent.

The effect of this is best shown by way of comparing to *--full-history* with parent rewriting. The example turns into:

```
  .-A---M---N---O
 /     /       /
I     B       D
 \   /       /
  `---------'
```

Note the major differences in *N*, *P*, and *Q* over *--full-history*:

- *N*'s parent list had *I* removed, because it is an ancestor of the other parent *M*. Still, *N* remained because it is !TREESAME.
- *P*'s parent list similarly had *I* removed. *P* was then removed completely, because it had one parent and is TREESAME.
- *Q*'s parent list had *Y* simplified to *X*. *X* was then removed, because it was a TREESAME root. *Q* was then removed completely, because it had one parent and is TREESAME.

There is another simplification mode available:

--ancestry-path[=<commit>]

Limit the displayed commits to those which are an ancestor of <commit>, or which are a descendant of <commit>, or are <commit> itself.

As an example use case, consider the following commit history:

```
    D---E-------F
   /     \       \
  B---C---G---H---I---J
 /                     \
A-------K---------------L--M
```

A regular *D..M* computes the set of commits that are ancestors of *M*, but excludes the ones that are ancestors of *D*. This is useful to see what happened to the history leading to *M* since *D*, in the sense that what does *M* have that did not exist in *D*. The result in this example would be all the commits, except *A* and *B* (and *D* itself, of course).

When we want to find out what commits in *M* are contaminated with the bug introduced by *D* and need fixing, however, we might want to view only the subset of *D..M* that are actually descendants of *D*, i.e. excluding *C* and *K*. This is exactly what the *--ancestry-path* option does. Applied to the *D..M* range, it results in:

```
E-------F
 \       \
  G---H---I---J
                \
                 L--M
```

We can also use *--ancestry-path=D* instead of *--ancestry-path* which means the same thing when applied to the *D..M* range but is just more explicit.

If we instead are interested in a given topic within this range, and all commits affected by that topic, we may only want to view the subset of *D..M* which contain that topic in their ancestry path. So, using *--ancestry-path=H D..M* for example would result in:

```
E
 \
  G---H---I---J
                \
                 L--M
```

Whereas *--ancestry-path=K D..M* would result in

```
K---------------L--M
```

Before discussing another option, *--show-pulls*, we need to create a new example history.

A common problem users face when looking at simplified history is that a commit they know changed a file somehow does not appear in the file's simplified history. Let's demonstrate a new example and show how options such as *--full-history* and *--simplify-merges* works in that case:

```
  .-A---M-----C--N---O---P
 /     / \  \  \/   /   /
I     B   \  R-'`-Z'   /
 \   /     \/         /
  \ /      /\        /
   `---X--'  `---Y--'
```

For this example, suppose *I* created *file.txt* which was modified by *A*, *B*, and *X* in different ways. The single-parent commits *C*, *Z*, and *Y* do not change *file.txt*. The merge commit *M* was created by resolving the merge conflict to include both changes from *A* and *B* and hence is not TREESAME to either. The merge commit *R*, however, was created by ignoring the contents of *file.txt* at *M* and taking only the contents of *file.txt* at *X*. Hence, *R* is TREESAME to *X* but not *M*. Finally, the natural merge resolution to create *N* is to take the contents of *file.txt* at *R*, so *N* is TREESAME to *R* but not *C*. The merge commits *O* and *P* are TREESAME to their first parents, but not to their second parents, *Z* and *Y* respectively.

When using the default mode, *N* and *R* both have a TREESAME parent, so those edges are walked and the others are ignored. The resulting history graph is:

```
I---X
```

When using *--full-history*, Git walks every edge. This will discover the commits *A* and *B* and the merge *M*, but also will reveal the merge commits *O* and *P*. With parent rewriting, the resulting graph is:

```
  .-A---M--------N---O---P
 /     / \  \  \/   /   /
I     B   \  R-'`--'   /
 \   /     \/         /
  \ /      /\        /
   `---X--'  `------'
```

Here, the merge commits *O* and *P* contribute extra noise, as they did not actually contribute a change to *file.txt*. They only merged a topic that was based on an older version of *file.txt*. This is a common issue in repositories using a workflow where many contributors work in parallel and merge their topic branches along a single trunk: many unrelated merges appear in the *--full-history* results.

When using the *--simplify-merges* option, the commits *O* and *P* disappear from the results. This is because the rewritten second parents of *O* and *P* are reachable from their first parents. Those edges are removed and then the

commits look like single-parent commits that are TREESAME to their parent. This also happens to the commit *N*, resulting in a history view as follows:

```
  .-A---M--.
 /     /    \
I     B      R
 \   /      /
  \ /      /
   `---X--'
```

In this view, we see all of the important single-parent changes from *A*, *B*, and *X*. We also see the carefully-resolved merge *M* and the not-so-carefully-resolved merge *R*. This is usually enough information to determine why the commits *A* and *B* "disappeared" from history in the default view. However, there are a few issues with this approach.

The first issue is performance. Unlike any previous option, the *--simplify-merges* option requires walking the entire commit history before returning a single result. This can make the option difficult to use for very large repositories.

The second issue is one of auditing. When many contributors are working on the same repository, it is important which merge commits introduced a change into an important branch. The problematic merge *R* above is not likely to be the merge commit that was used to merge into an important branch. Instead, the merge *N* was used to merge *R* and *X* into the important branch. This commit may have information about why the change *X* came to override the changes from *A* and *B* in its commit message.

--show-pulls

In addition to the commits shown in the default history, show each merge commit that is not TREESAME to its first parent but is TREESAME to a later parent.

When a merge commit is included by *--show-pulls*, the merge is treated as if it "pulled" the change from another branch. When using *--show-pulls* on this example (and no other options) the resulting graph is:

```
I---X---R---N
```

Here, the merge commits *R* and *N* are included because they pulled the commits *X* and *R* into the base branch, respectively. These merges are the reason the commits *A* and *B* do not appear in the default history.

When *--show-pulls* is paired with *--simplify-merges*, the graph includes all of the necessary information:

```
  .-A---M--.   N
 /     /    \ /
I     B      R
 \   /      /
  \ /      /
   `---X--'
```

Notice that since *M* is reachable from *R*, the edge from *N* to *M* was simplified away. However, *N* still appears in the history as an important commit because it "pulled" the change *R* into the main branch.

The *--simplify-by-decoration* option allows you to view only the big picture of the topology of the history, by omitting commits that are not referenced by tags. Commits are marked as !TREESAME (in other words, kept after history simplification rules described above) if (1) they are referenced by tags, or (2) they change the contents of the paths given on the command line. All other commits are marked as TREESAME (subject to be simplified away).

3. Commit Ordering

By default, the commits are shown in reverse chronological order.

--date-order

Show no parents before all of its children are shown, but otherwise show commits in the commit timestamp order.

--author-date-order

Show no parents before all of its children are shown, but otherwise show commits in the author timestamp order.

--topo-order

Show no parents before all of its children are shown, and avoid showing commits on multiple lines of history intermixed.

For example, in a commit history like this:

```
---1----2----4----7
    \               \
     3----5----6----8---
```

where the numbers denote the order of commit timestamps, *git rev-list* and friends with *--date-order* show the commits in the timestamp order: 8 7 6 5 4 3 2 1.

With *--topo-order*, they would show 8 6 5 3 7 4 2 1 (or 8 7 4 2 6 5 3 1); some older commits are shown before newer ones in order to avoid showing the commits from two parallel development track mixed together.

--reverse

Output the commits chosen to be shown (see Commit Limiting section above) in reverse order. Cannot be combined with *--walk-reflogs*.

4. Object Traversal

These options are mostly targeted for packing of Git repositories.

--no-walk[=(sorted|unsorted)]

Only show the given commits, but do not traverse their ancestors. This has no effect if a range is specified. If the argument *unsorted* is given, the commits are shown in the order they were given on the command line. Otherwise (if *sorted* or no argument was given), the commits are shown in reverse chronological order by commit time. Cannot be combined with *--graph*.

--do-walk

Overrides a previous *--no-walk*.

5. Commit Formatting

--pretty[=<format>] , --format=<format>

Pretty-print the contents of the commit logs in a given format, where *<format>* can be one of *oneline*, *short*, *medium*, *full*, *fuller*, *reference*, *email*, *raw*, *format:<string>* and *tformat:<string>*. When *<format>* is none of the above, and has *%placeholder* in it, it acts as if *--pretty=tformat:<format>* were given.

See the "PRETTY FORMATS" section for some additional details for each format. When *=<format>* part is omitted, it defaults to *medium*.

Note: you can specify the default pretty format in the repository configuration (see Section G.3.29, “git-config(1)”).

--abbrev-commit

Instead of showing the full 40-byte hexadecimal commit object name, show a prefix that names the object uniquely. "--abbrev=<n>" (which also modifies diff output, if it is displayed) option can be used to specify the minimum length of the prefix.

This should make "--pretty=oneline" a whole lot more readable for people using 80-column terminals.

--no-abbrev-commit

Show the full 40-byte hexadecimal commit object name. This negates *--abbrev-commit*, either explicit or implied by other options such as "--oneline". It also overrides the *log.abbrevCommit* variable.

--oneline

This is a shorthand for "--pretty=oneline --abbrev-commit" used together.

--encoding=<encoding>

Commit objects record the character encoding used for the log message in their encoding header; this option can be used to tell the command to re-code the commit log message in the encoding preferred by the user. For non plumbing commands this defaults to UTF-8. Note that if an object claims to be encoded in *X* and we are outputting in *X*, we will output the object verbatim; this means that invalid sequences in the original commit may be copied to the output. Likewise, if iconv(3) fails to convert the commit, we will quietly output the original object verbatim.

--expand-tabs=<n> , --expand-tabs , --no-expand-tabs

Perform a tab expansion (replace each tab with enough spaces to fill to the next display column that is a multiple of <*n*>) in the log message before showing it in the output. *--expand-tabs* is a short-hand for *--expand-tabs=8*, and *--no-expand-tabs* is a short-hand for *--expand-tabs=0*, which disables tab expansion.

By default, tabs are expanded in pretty formats that indent the log message by 4 spaces (i.e. *medium*, which is the default, *full*, and *fuller*).

--notes[=<ref>]

Show the notes (see Section G.3.94, “git-notes(1)”) that annotate the commit, when showing the commit log message. This is the default for *git log*, *git show* and *git whatchanged* commands when there is no *--pretty*, *--format*, or *--oneline* option given on the command line.

By default, the notes shown are from the notes refs listed in the *core.notesRef* and *notes.displayRef* variables (or corresponding environment overrides). See Section G.3.29, “git-config(1)” for more details.

With an optional <*ref*> argument, use the ref to find the notes to display. The ref can specify the full refname when it begins with *refs/notes/*; when it begins with *notes/*, *refs/* and otherwise *refs/notes/* is prefixed to form the full name of the ref.

Multiple --notes options can be combined to control which notes are being displayed. Examples: "--notes=foo" will show only notes from "refs/notes/foo"; "--notes=foo --notes" will show both notes from "refs/notes/foo" and from the default notes ref(s).

--no-notes

Do not show notes. This negates the above *--notes* option, by resetting the list of notes refs from which notes are shown. Options are parsed in the order given on the command line, so e.g. "--notes --notes=foo --no-notes --notes=bar" will only show notes from "refs/notes/bar".

--show-notes-by-default

Show the default notes unless options for displaying specific notes are given.

--show-notes[=<ref>] , --[no-]standard-notes

These options are deprecated. Use the above --notes/--no-notes options instead.

--show-signature

Check the validity of a signed commit object by passing the signature to *gpg --verify* and show the output.

--relative-date

Synonym for *--date=relative*.

--date=<format>

Only takes effect for dates shown in human-readable format, such as when using *--pretty*. *log.date* config variable sets a default value for the log command's *--date* option. By default, dates are shown in the original time zone (either committer's or author's). If *-local* is appended to the format (e.g., *iso-local*), the user's local time zone is used instead.

--date=relative shows dates relative to the current time, e.g. 2 hours ago. The *-local* option has no effect for *--date=relative*.

--date=local is an alias for *--date=default-local*.

--date=iso (or *--date=iso8601*) shows timestamps in a ISO 8601-like format. The differences to the strict ISO 8601 format are:

- a space instead of the *T* date/time delimiter
- a space between time and time zone
- no colon between hours and minutes of the time zone

--date=iso-strict (or *--date=iso8601-strict*) shows timestamps in strict ISO 8601 format.

--date=rfc (or *--date=rfc2822*) shows timestamps in RFC 2822 format, often found in email messages.

--date=short shows only the date, but not the time, in *YYYY-MM-DD* format.

--date=raw shows the date as seconds since the epoch (1970-01-01 00:00:00 UTC), followed by a space, and then the timezone as an offset from UTC (a + or - with four digits; the first two are hours, and the second two are minutes). I.e., as if the timestamp were formatted with *strftime("%s %z")*). Note that the *-local* option does not affect the seconds-since-epoch value (which is always measured in UTC), but does switch the accompanying timezone value.

--date=human shows the timezone if the timezone does not match the current time-zone, and doesn't print the whole date if that matches (ie skip printing year for dates that are "this year", but also skip the whole date itself if it's in the last few days and we can just say what weekday it was). For older dates the hour and minute is also omitted.

--date=unix shows the date as a Unix epoch timestamp (seconds since 1970). As with *--raw*, this is always in UTC and therefore *-local* has no effect.

--date=format:... feeds the format ... to your system *strftime*, except for %s, %z, and %Z, which are handled internally. Use *--date=format:%c* to show the date in your system locale's preferred format. See the *strftime* manual for a complete list of format placeholders. When using *-local*, the correct syntax is *--date=format-local:...*.

--date=default is the default format, and is based on ctime(3) output. It shows a single line with three-letter day of the week, three-letter month, day-of-month, hour-minute-seconds in "HH:MM:SS" format, followed by 4-digit year, plus timezone information, unless the local time zone is used, e.g. *Thu Jan 1 00:00:00 1970 +0000*.

--parents

Print also the parents of the commit (in the form "commit parent…"). Also enables parent rewriting, see *History Simplification* above.

--children

Print also the children of the commit (in the form "commit child…"). Also enables parent rewriting, see *History Simplification* above.

--left-right

Mark which side of a symmetric difference a commit is reachable from. Commits from the left side are prefixed with < and those from the right with >. If combined with *--boundary*, those commits are prefixed with -.

For example, if you have this topology:

```
                y---b---b  branch B
               / \ /
              /   .
             /   / \
           o---x---a---a  branch A
```

you would get an output like this:

```
$ git rev-list --left-right --boundary --pretty=oneline A...B

>bbbbbbb... 3rd on b
>bbbbbbb... 2nd on b
<aaaaaaa... 3rd on a
<aaaaaaa... 2nd on a
-yyyyyyy... 1st on b
-xxxxxxx... 1st on a
```

--graph

Draw a text-based graphical representation of the commit history on the left hand side of the output. This may cause extra lines to be printed in between commits, in order for the graph history to be drawn properly. Cannot be combined with *--no-walk*.

This enables parent rewriting, see *History Simplification* above.

This implies the *--topo-order* option by default, but the *--date-order* option may also be specified.

--show-linear-break[=<barrier>]

When --graph is not used, all history branches are flattened which can make it hard to see that the two consecutive commits do not belong to a linear branch. This option puts a barrier in between them in that case. If *<barrier>* is specified, it is the string that will be shown instead of the default one.

PRETTY FORMATS

If the commit is a merge, and if the pretty-format is not *oneline*, *email* or *raw*, an additional line is inserted before the *Author:* line. This line begins with "Merge: " and the hashes of ancestral commits are printed, separated by spaces. Note that the listed commits may not necessarily be the list of the **direct** parent commits if you have limited your view of history: for example, if you are only interested in changes related to a certain directory or file.

There are several built-in formats, and you can define additional formats by setting a pretty.<name> config option to either another format name, or a *format:* string, as described below (see Section G.3.29, “git-config(1)”). Here are the details of the built-in formats:

- *oneline*

  ```
  <hash> <title-line>
  ```

  This is designed to be as compact as possible.

- *short*

```
commit <hash>
Author: <author>

<title-line>
```

- *medium*

```
commit <hash>
Author: <author>
Date:   <author-date>

<title-line>

<full-commit-message>
```

- *full*

```
commit <hash>
Author: <author>
Commit: <committer>

<title-line>

<full-commit-message>
```

- *fuller*

```
commit <hash>
Author:     <author>
AuthorDate: <author-date>
Commit:     <committer>
CommitDate: <committer-date>

<title-line>

<full-commit-message>
```

- *reference*

```
<abbrev-hash> (<title-line>, <short-author-date>)
```

This format is used to refer to another commit in a commit message and is the same as *--pretty='format:%C(auto)%h (%s, %ad)'*. By default, the date is formatted with *--date=short* unless another *--date* option is explicitly specified. As with any *format:* with format placeholders, its output is not affected by other options like *--decorate* and *--walk-reflogs*.

- *email*

```
From <hash> <date>
From: <author>
Date: <author-date>
Subject: [PATCH] <title-line>

<full-commit-message>
```

- *mboxrd*

Like *email*, but lines in the commit message starting with "From " (preceded by zero or more ">") are quoted with ">" so they aren't confused as starting a new commit.

- *raw*

The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the hashes are displayed in full, regardless of whether --abbrev or --no-abbrev are used, and *parents* information show the true parent commits, without taking grafts or history simplification into account. Note that this format affects the way commits are displayed, but not the way the diff is shown e.g. with *git log --raw*. To get full object names in a raw diff format, use *--no-abbrev*.

- *format:<format-string>*

  The *format:<format-string>* format allows you to specify which information you want to show. It works a little bit like printf format, with the notable exception that you get a newline with *%n* instead of *\n*.

  E.g, *format:"The author of %h was %an, %ar%nThe title was >>%s<<%n"* would show something like this:

```
The author of fe6e0ee was Junio C Hamano, 23 hours ago
The title was >>t4119: test autocomputing -p<n> for traditional diff
 input.<<
```

  The placeholders are:

  - Placeholders that expand to a single literal character:

    %n

    newline

    %%

    a raw *%*

    %x00

    %x followed by two hexadecimal digits is replaced with a byte with the hexadecimal digits' value (we will call this "literal formatting code" in the rest of this document).

  - Placeholders that affect formatting of later placeholders:

    %Cred

    switch color to red

    %Cgreen

    switch color to green

    %Cblue

    switch color to blue

    %Creset

    reset color

    %C(…)

    color specification, as described under Values in the "CONFIGURATION FILE" section of Section G.3.29, “git-config(1)”. By default, colors are shown only when enabled for log output (by *color.diff*, *color.ui*, or *--color*, and respecting the *auto* settings of the former if we are going to a terminal). *%C(auto,...)* is accepted as a historical synonym for the default (e.g., *%C(auto,red)*). Specifying *%C(always,...)* will show the colors even when color is not otherwise enabled (though consider just using *--color=always* to enable color for the whole output, including this format and anything else git might color). *auto* alone (i.e. *%C(auto)*) will turn on auto coloring on the next placeholders until the color is switched again.

%m

left (<), right (>) or boundary (-) mark

%w([<w>[,<i1>[,<i2>]]])

switch line wrapping, like the -w option of Section G.3.130, “git-shortlog(1)”.

%<(<N> [,trunc|ltrunc|mtrunc])

make the next placeholder take at least N column widths, padding spaces on the right if necessary. Optionally truncate (with ellipsis ..) at the left (ltrunc) *..ft*, the middle (mtrunc) *mi..le*, or the end (trunc) *rig..*, if the output is longer than N columns. Note 1: that truncating only works correctly with N >= 2. Note 2: spaces around the N and M (see below) values are optional. Note 3: Emojis and other wide characters will take two display columns, which may over-run column boundaries. Note 4: decomposed character combining marks may be misplaced at padding boundaries.

%<|(<M>)

make the next placeholder take at least until Mth display column, padding spaces on the right if necessary. Use negative M values for column positions measured from the right hand edge of the terminal window.

%>(<N>), %>|(<M>)

similar to *%<(<N>), %<|(<M>)* respectively, but padding spaces on the left

%>>(<N>), %>>|(<M>)

similar to *%>(<N>), %>|(<M>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces

%><(<N>), %><|(<M>)

similar to *%<(<N>), %<|(<M>)* respectively, but padding both sides (i.e. the text is centered)

- Placeholders that expand to information extracted from the commit:

%H

commit hash

%h

abbreviated commit hash

%T

tree hash

%t

abbreviated tree hash

%P

parent hashes

%p

abbreviated parent hashes

%an

author name

%aN

author name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ae

author email

%aE

author email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%al

author email local-part (the part before the @ sign)

%aL

author local-part (see *%al*) respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ad

author date (format respects --date= option)

%aD

author date, RFC2822 style

%ar

author date, relative

%at

author date, UNIX timestamp

%ai

author date, ISO 8601-like format

%aI

author date, strict ISO 8601 format

%as

author date, short format (*YYYY-MM-DD*)

%ah

author date, human style (like the *--date=human* option of Section G.3.120, “git-rev-list(1)”)

%cn

committer name

%cN

committer name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ce

committer email

%cE

committer email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%cl

committer email local-part (the part before the @ sign)

%cL

committer local-part (see *%cl*) respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%cd

committer date (format respects --date= option)

%cD

committer date, RFC2822 style

%cr

committer date, relative

%ct

committer date, UNIX timestamp

%ci

committer date, ISO 8601-like format

%cI

committer date, strict ISO 8601 format

%cs

committer date, short format (*YYYY-MM-DD*)

%ch

committer date, human style (like the *--date=human* option of Section G.3.120, “git-rev-list(1)”)

%d

ref names, like the --decorate option of Section G.3.74, “git-log(1)”

%D

ref names without the " (", ")" wrapping.

%(decorate[:<options>])

ref names with custom decorations. The *decorate* string may be followed by a colon and zero or more comma-separated options. Option values may contain literal formatting codes. These must be used for commas (*%x2C*) and closing parentheses (*%x29*), due to their role in the option syntax.

- *prefix=<value>*: Shown before the list of ref names. Defaults to " *(*".
- *suffix=<value>*: Shown after the list of ref names. Defaults to "*)*".
- *separator=<value>*: Shown between ref names. Defaults to ", ".
- *pointer=<value>*: Shown between HEAD and the branch it points to, if any. Defaults to " -> ".
- *tag=<value>*: Shown before tag names. Defaults to "*tag:* ".

For example, to produce decorations with no wrapping or tag annotations, and spaces as separators:

%(decorate:prefix=,suffix=,tag=,separator=)

%(describe[:<options>])

human-readable name, like Section G.3.39, “git-describe(1)”; empty string for undescribable commits. The *describe* string may be followed by a colon and zero or more comma-separated options. Descriptions can be inconsistent when tags are added or removed at the same time.

- *tags[=<bool-value>]*: Instead of only considering annotated tags, consider lightweight tags as well.
- *abbrev=<number>*: Instead of using the default number of hexadecimal digits (which will vary according to the number of objects in the repository with a default of 7) of the abbreviated object name, use <number> digits, or as many digits as needed to form a unique object name.
- *match=<pattern>*: Only consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix.
- *exclude=<pattern>*: Do not consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix.

%S

ref name given on the command line by which the commit was reached (like *git log --source*), only works with *git log*

%e

encoding

%s

subject

%f

sanitized subject line, suitable for a filename

%b

body

%B

raw body (unwrapped subject and body)

%N

commit notes

%GG

raw verification message from GPG for a signed commit

%G?

show "G" for a good (valid) signature, "B" for a bad signature, "U" for a good signature with unknown validity, "X" for a good signature that has expired, "Y" for a good signature made by an expired key, "R" for a good signature made by a revoked key, "E" if the signature cannot be checked (e.g. missing key) and "N" for no signature

%GS

show the name of the signer for a signed commit

%GK

show the key used to sign a signed commit

%GF

show the fingerprint of the key used to sign a signed commit

%GP

show the fingerprint of the primary key whose subkey was used to sign a signed commit

%GT

show the trust level for the key used to sign a signed commit

%gD

reflog selector, e.g., *refs/stash@{1}* or *refs/stash@{2 minutes ago}*; the format follows the rules described for the *-g* option. The portion before the @ is the refname as given on the command line (so *git log -g refs/heads/master* would yield *refs/heads/master@{0}*).

%gd

shortened reflog selector; same as *%gD*, but the refname portion is shortened for human readability (so *refs/heads/master* becomes just *master*).

%gn

reflog identity name

%gN

reflog identity name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ge

reflog identity email

%gE

reflog identity email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%gs

reflog subject

%(trailers[:<options>])

display the trailers of the body as interpreted by Section G.3.73, “git-interpret-trailers(1)”. The *trailers* string may be followed by a colon and zero or more comma-separated options. If any option is provided multiple times, the last occurrence wins.

- *key=<key>*: only show trailers with specified <key>. Matching is done case-insensitively and trailing colon is optional. If option is given multiple times trailer lines matching any of the keys are shown. This option automatically enables the *only* option so that non-trailer lines in the trailer block are hidden. If that is not desired it can be disabled with *only=false*. E.g., *%(trailers:key=Reviewed-by)* shows trailer lines with key *Reviewed-by*.
- *only[=<bool>]*: select whether non-trailer lines from the trailer block should be included.
- *separator=<sep>*: specify the separator inserted between trailer lines. Defaults to a line feed character. The string <sep> may contain the literal formatting codes described above. To use comma as separator one must use *%x2C* as it would otherwise be parsed as next option. E.g., *%(trailers:key=Ticket,separator=%x2C)* shows all trailer lines whose key is "Ticket" separated by a comma and a space.
- *unfold[=<bool>]*: make it behave as if interpret-trailer's *--unfold* option was given. E.g., *%(trailers:only,unfold=true)* unfolds and shows all trailer lines.
- *keyonly[=<bool>]*: only show the key part of the trailer.
- *valueonly[=<bool>]*: only show the value part of the trailer.
- *key_value_separator=<sep>*: specify the separator inserted between the key and value of each trailer. Defaults to ": ". Otherwise it shares the same semantics as *separator=<sep>* above.

> **Note**
>
> Some placeholders may depend on other options given to the revision traversal engine. For example, the *%g** reflog options will insert an empty string unless we are traversing reflog entries (e.g., by *git log -g*). The *%d* and *%D* placeholders will use the "short" decoration format if *--decorate* was not already provided on the command line.

The boolean options accept an optional value *[=<bool-value>]*. The values *true*, *false*, *on*, *off* etc. are all accepted. See the "boolean" sub-section in "EXAMPLES" in Section G.3.29, “git-config(1)”. If a boolean option is given with no value, it's enabled.

If you add a + (plus sign) after % of a placeholder, a line-feed is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

If you add a - (minus sign) after % of a placeholder, all consecutive line-feeds immediately preceding the expansion are deleted if and only if the placeholder expands to an empty string.

If you add a ` ` (space) after % of a placeholder, a space is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

- *tformat:*

  The *tformat:* format works exactly like *format:*, except that it provides "terminator" semantics instead of "separator" semantics. In other words, each commit has the message terminator character (usually a newline) appended, rather than a separator placed between entries. This means that the final entry of a single-line format will be properly terminated with a new line, just as the "oneline" format does. For example:

  ```
  $ git log -2 --pretty=format:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973 -- NO NEWLINE

  $ git log -2 --pretty=tformat:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973
  ```

  In addition, any unrecognized string that has a *%* in it is interpreted as if it has *tformat:* in front of it. For example, these two are equivalent:

  ```
  $ git log -2 --pretty=tformat:%h 4da45bef
  $ git log -2 --pretty=%h 4da45bef
  ```

DIFF FORMATTING

By default, *git log* does not generate any diff output. The options below can be used to show the changes made by each commit.

Note that unless one of *--diff-merges* variants (including short *-m*, *-c*, *--cc*, and *--dd* options) is explicitly given, merge commits will not show a diff, even if a diff format like *--patch* is selected, nor will they match search options like *-S*. The exception is when *--first-parent* is in use, in which case *first-parent* is the default format for merge commits.

-p , -u , --patch

Generate patch (see the section called “Generating patch text with -p”).

-s , --no-patch

Suppress all output from the diff machinery. Useful for commands like *git show* that show the patch by default to squelch their output, or to cancel the effect of options like *--patch*, *--stat* earlier on the command line in an alias.

-m

Show diffs for merge commits in the default format. This is similar to *--diff-merges=on*, except *-m* will produce no output unless *-p* is given as well.

-c

Produce combined diff output for merge commits. Shortcut for *--diff-merges=combined -p*.

--cc

Produce dense combined diff output for merge commits. Shortcut for *--diff-merges=dense-combined -p*.

--dd

Produce diff with respect to first parent for both merge and regular commits. Shortcut for *--diff-merges=first-parent -p*.

--remerge-diff

Produce remerge-diff output for merge commits. Shortcut for *--diff-merges=remerge -p*.

--no-diff-merges

Synonym for *--diff-merges=off*.

--diff-merges=<format>

Specify diff format to be used for merge commits. Default is *off* unless *--first-parent* is in use, in which case *first-parent* is the default.

The following formats are supported:

off, none

Disable output of diffs for merge commits. Useful to override implied value.

on, m

Make diff output for merge commits to be shown in the default format. The default format can be changed using *log.diffMerges* configuration variable, whose default value is *separate*.

first-parent, 1

Show full diff with respect to first parent. This is the same format as *--patch* produces for non-merge commits.

separate

Show full diff with respect to each of parents. Separate log entry and diff is generated for each parent.

combined, c

Show differences from each of the parents to the merge result simultaneously instead of showing pairwise diff between a parent and the result one at a time. Furthermore, it lists only files which were modified from all parents.

dense-combined, cc

Further compress output produced by *--diff-merges=combined* by omitting uninteresting hunks whose contents in the parents have only two variants and the merge result picks one of them without modification.

remerge, r

Remerge two-parent merge commits to create a temporary tree object--potentially containing files with conflict markers and such. A diff is then shown between that temporary tree and the actual merge commit.

The output emitted when this option is used is subject to change, and so is its interaction with other options (unless explicitly documented).

--combined-all-paths

This flag causes combined diffs (used for merge commits) to list the name of the file from all parents. It thus only has effect when *--diff-merges=[dense-]combined* is in use, and is likely only useful if filename changes are detected (i.e. when either rename or copy detection have been requested).

-U<n> , --unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies *--patch*.

--output=<file>

Output to a specific file instead of stdout.

--output-indicator-new=<char> , --output-indicator-old=<char> , --output-indicator-context=<char>

Specify the character used to indicate new, old or context lines in the generated patch. Normally they are +, - and ' ' respectively.

--raw

For each commit, show a summary of changes using the raw diff format. See the "RAW OUTPUT FORMAT" section of Section G.3.44, “git-diff(1)”. This is different from showing the log itself in raw format, which you can achieve with *--format=raw*.

--patch-with-raw

Synonym for *-p --raw*.

-t

Show the tree objects in the diff output.

--indent-heuristic

Enable the heuristic that shifts diff hunk boundaries to make patches easier to read. This is the default.

--no-indent-heuristic

Disable the indent heuristic.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--anchored=<text>

Generate a diff using the "anchored diff" algorithm.

This option may be specified more than once.

If a line exists in both the source and destination, exists only once, and starts with this text, this algorithm attempts to prevent it from appearing as a deletion or addition in the output. It uses the "patience diff" algorithm internally.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

default, *myers*

The basic greedy diff algorithm. Currently, this is the default.

minimal

Spend extra time to make sure the smallest possible diff is produced.

patience

Use "patience diff" algorithm when generating patches.

histogram

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured the *diff.algorithm* variable to a non-default value and want to use the default one, then you have to use *--diff-algorithm=default* option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by *<width>*. The width of the filename part can be limited by giving another width *<name-width>* after a comma or by setting *diff.statNameWidth=<width>*. The width of the graph part can be limited by using *--stat-graph-width=<width>* or by setting *diff.statGraphWidth=<width>*. Using *--stat* or *--stat-graph-width* affects all commands generating a stat graph, while setting *diff.statNameWidth* or *diff.statGraphWidth* does not affect *git format-patch*. By giving a third parameter *<count>*, you can limit the output to the first *<count>* lines, followed by ... if there are more.

These parameters can also be set individually with *--stat-width=<width>*, *--stat-name-width=<name-width>* and *--stat-count=<count>*.

--compact-summary

Output a condensed summary of extended header information such as file creations or deletions ("new" or "gone", optionally "+l" if it's a symlink) and mode changes ("+x" or "-x" for adding or removing executable bit respectively) in diffstat. The information is put between the filename part and the graph part. Implies *--stat*.

--numstat

Similar to *--stat*, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two - instead of saying *0 0*.

--shortstat

Output only the last line of the *--stat* format containing total number of modified files, as well as number of added and deleted lines.

-X[<param1,param2,…>] , --dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of *--dirstat* can be customized by passing it a comma separated list of parameters. The defaults are controlled by the *diff.dirstat* configuration variable (see Section G.3.29, “git-config(1)”). The following parameters are available:

changes

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

lines

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive *--dirstat* behavior than the *changes* behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other *--*stat* options.

files

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest *--dirstat* behavior, since it does not have to look at the file contents at all.

cumulative

Count changes in a child directory for the parent directory as well. Note that when using *cumulative*, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the *noncumulative* parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: *--dirstat=files,10,cumulative*.

--cumulative

Synonym for --dirstat=cumulative

--dirstat-by-file[=<param1,param2>…]

Synonym for --dirstat=files,<param1>,<param2>…

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for *-p --stat*.

-z

Separate the commits with NULs instead of newlines.

Also, when *--raw* or *--numstat* has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

--name-only

Show only the name of each changed file in the post-image tree. The file names are often encoded in UTF-8. For more information see the discussion about encoding in the Section G.3.74, “git-log(1)” manual page.

--name-status

Show only the name(s) and status of each changed file. See the description of the *--diff-filter* option on what the status letters mean. Just like *--name-only* the file names are often encoded in UTF-8.

--submodule[=<format>]

Specify how differences in submodules are shown. When specifying *--submodule=short* the *short* format is used. This format just shows the names of the commits at the beginning and end of the range. When *--submodule* or *--submodule=log* is specified, the *log* format is used. This format lists the commits in the range

like Section G.3.141, “git-submodule(1)” *summary* does. When *--submodule=diff* is specified, the *diff* format is used. This format shows an inline diff of the changes in the submodule contents between the commit range. Defaults to *diff.submodule* or the *short* format if the config option is unset.

--color[=<when>]

Show colored diff. *--color* (i.e. without =*<when>*) is the same as *--color=always*. *<when>* can be one of *always*, *never*, or *auto*.

--no-color

Turn off colored diff. It is the same as *--color=never*.

--color-moved[=<mode>]

Moved lines of code are colored differently. The <mode> defaults to *no* if the option is not given and to *zebra* if the option with no mode is given. The mode must be one of:

no

Moved lines are not highlighted.

default

Is a synonym for *zebra*. This may change to a more sensible mode in the future.

plain

Any line that is added in one location and was removed in another location will be colored with *color.diff.newMoved*. Similarly *color.diff.oldMoved* will be used for removed lines that are added somewhere else in the diff. This mode picks up any moved line, but it is not very useful in a review to determine if a block of code was moved without permutation.

blocks

Blocks of moved text of at least 20 alphanumeric characters are detected greedily. The detected blocks are painted using either the *color.diff.{old,new}Moved* color. Adjacent blocks cannot be told apart.

zebra

Blocks of moved text are detected as in *blocks* mode. The blocks are painted using either the *color.diff.{old,new}Moved* color or *color.diff.{old,new}MovedAlternative*. The change between the two colors indicates that a new block was detected.

dimmed-zebra

Similar to *zebra*, but additional dimming of uninteresting parts of moved code is performed. The bordering lines of two adjacent blocks are considered interesting, the rest is uninteresting. *dimmed_zebra* is a deprecated synonym.

--no-color-moved

Turn off move detection. This can be used to override configuration settings. It is the same as *--color-moved=no*.

--color-moved-ws=<modes>

This configures how whitespace is ignored when performing the move detection for *--color-moved*. These modes can be given as a comma separated list:

no

Do not ignore whitespace when performing move detection.

ignore-space-at-eol

Ignore changes in whitespace at EOL.

ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

allow-indentation-change

Initially ignore any whitespace in the move detection, then group the moved code blocks only into a block if the change in whitespace is the same per line. This is incompatible with the other modes.

--no-color-moved-ws

Do not ignore whitespace when performing move detection. This can be used to override configuration settings. It is the same as *--color-moved-ws=no*.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see *--word-diff-regex* below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies *--color*.

plain

Show words as *[-removed-]* and *{+added+}*. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a +/-/` ` character at the beginning of the line and extending to the end of the line. Newlines in the input are represented by a tilde ~ on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies *--word-diff* unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append *|[^[:space:]]* to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, *--word-diff-regex=.* will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see Section G.4.2, “gitattributes(5)” or Section G.3.29, “git-config(1)”. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to *--word-diff=color* plus (if a regex was specified) *--word-diff-regex=<regex>*.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--[no-]rename-empty

Whether to use empty blobs as rename source.

--check

Warn if changes introduce conflict markers or whitespace errors. What are considered whitespace errors is controlled by *core.whitespace* configuration. By default, trailing whitespaces (including lines that consist solely of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors in the *context*, *old* or *new* lines of the diff. Multiple values are separated by comma, *none* resets previous values, *default* reset the list to *new* and *all* is a shorthand for *old,new,context*. When this option is not given, and the configuration variable *diff.wsErrorHighlight* is not set, only whitespace errors in *new* lines are highlighted. The whitespace errors are colored with *color.diff.whitespace*.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to *--full-index*, output a binary diff that can be applied with *git-apply*. Implies *--patch*.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show the shortest prefix that is at least *<n>* hexdigits long that uniquely refers the object. In diff-patch output format, *--full-index* takes higher precedence, i.e. if *--full-index* is specified, full blob names will be shown regardless of *--abbrev*. Non default number of digits can be specified with *--abbrev=<n>*.

-B[<n>][/<m>] , --break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything new, and the number *m* controls this aspect of the -B option (defaults to 60%). *-B/70%* specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number *n* controls this aspect of the -B option (defaults to 50%). *-B20%* specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>] , --find-renames[=<n>]

If generating diffs, detect and report renames for each commit. For following files across renames while traversing history, see *--follow*. If *n* is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, *-M90%* means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a *%* sign, the number is to be read as a fraction, with a decimal point before it. I.e., *-M5* becomes 0.5, and is thus the same as *-M50%*. Similarly, *-M05* is the same as *-M5%*. To limit detection to exact renames, use *-M100%*. The default similarity index is 50%.

-C[<n>] , --find-copies[=<n>]

Detect copies as well as renames. See also *--find-copies-harder*. If *n* is specified, it has the same meaning as for *-M<n>*.

--find-copies-harder

For performance reasons, by default, *-C* option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one *-C* option has the same effect.

-D , --irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and */dev/null*. The resulting patch is not meant to be applied with *patch* or *git apply*; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lacks enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with *-B*, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The *-M* and *-C* options involve some preliminary steps that can detect subsets of renames/copies cheaply, followed by an exhaustive fallback portion that compares all remaining unpaired destinations to all relevant sources. (For renames, only remaining unpaired sources are relevant; for copies, all original sources are relevant.) For N sources and destinations, this exhaustive check is O(N^2). This option prevents the exhaustive portion of rename/copy detection from running if the number of source/destination files involved exceeds the specified number. Defaults to diff.renameLimit. Note that a value of 0 is treated as unlimited.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (*A*), Copied (*C*), Deleted (*D*), Modified (*M*), Renamed (*R*), have their type (i.e. regular file, symlink, submodule, …) changed (*T*), are Unmerged (*U*), are Unknown (*X*), or have had their pairing Broken (*B*). Any combination of the filter characters (including none) can be used. When * (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

Also, these upper-case letters can be downcased to exclude. E.g. *--diff-filter=ad* excludes added and deleted paths.

Note that not all diffs can feature all types. For instance, copied and renamed entries cannot appear if detection for those types is disabled.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter's use.

It is useful when you're looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into *-S*, and keep going until you get the very first version of the block.

Binary files are searched as well.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between *-S<regex> --pickaxe-regex* and *-G<regex>*, consider a commit with the following diff in the same file:

```
+    return frotz(nitfol, two->ptr, 1, 0);
...
-    hit = frotz(nitfol, mf2.ptr, 1, 0);
```

While *git log -G"frotz\(nitfol"* will show this commit, *git log -S"frotz\(nitfol" --pickaxe-regex* will not (because the number of occurrences of that string did not change).

Unless *--text* is supplied patches of binary files without a textconv filter will be ignored.

See the *pickaxe* entry in Section G.4.4, “gitdiffcore(7)” for more information.

--find-object=<object-id>

Look for differences that change the number of occurrences of the specified object. Similar to *-S*, just the argument is different in that it doesn't search for a specific string but for a specific object id.

The object can be a blob or a submodule commit. It implies the *-t* option in *git-log* to also find trees.

--pickaxe-all

When *-S* or *-G* finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to *-S* as an extended POSIX regular expression to match.

-O<orderfile>

Control the order in which files appear in the output. This overrides the *diff.orderFile* configuration variable (see Section G.3.29, “git-config(1)”). To cancel *diff.orderFile*, use *-O/dev/null*.

The output order is determined by the order of glob patterns in <orderfile>. All files with pathnames that match the first pattern are output first, all files with pathnames that match the second pattern (but not the first) are output next, and so on. All files with pathnames that do not match any pattern are output last, as if there was an implicit match-all pattern at the end of the file. If multiple pathnames have the same rank (they match the same pattern but no earlier patterns), their output order relative to each other is the normal order.

<orderfile> is parsed as follows:

- Blank lines are ignored, so they can be used as separators for readability.
- Lines starting with a hash ("#") are ignored, so they can be used for comments. Add a backslash ("\") to the beginning of the pattern if it starts with a hash.
- Each other line contains a single pattern.

Patterns have the same syntax and semantics as patterns used for fnmatch(3) without the FNM_PATHNAME flag, except a pathname also matches a pattern if removing any number of the final pathname components matches the pattern. For example, the pattern "*foo*bar*" matches "*fooasdfbar*" and "*foo/bar/baz/asdf*" but not "*foobarx*".

--skip-to=<file> , --rotate-to=<file>

Discard the files before the named <file> from the output (i.e. *skip to*), or move them to the end of the output (i.e. *rotate to*). These options were invented primarily for the use of the *git difftool* command, and may not be very useful otherwise.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>] , --no-relative

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument. *--no-relative* can be used to countermand both *diff.relative* config option and previous *--relative*.

-a , --text

Treat all files as text.

--ignore-cr-at-eol

Ignore carriage-return at the end of line when doing a comparison.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b , --ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w , --ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

-I<regex> , --ignore-matching-lines=<regex>

Ignore changes whose all lines match <regex>. This option may be specified more than once.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other. Defaults to *diff.interHunkContext* or 0 if the config option is unset.

-W , --function-context

Show whole function as context lines for each change. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with Section G.4.2, “gitattributes(5)”, you need to use this option with Section G.3.74, “git-log(1)” and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv , --no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See Section G.4.2, “gitattributes(5)” for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for Section G.3.44, “git-diff(1)” and Section G.3.74, “git-log(1)”, but not for Section G.3.54, “git-format-patch(1)” or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in Section G.3.29, “git-config(1)” or Section G.4.10, “gitmodules(5)”. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

--default-prefix

Use the default source and destination prefixes ("a/" and "b/"). This overrides configuration variables such as *diff.noprefix*, *diff.srcPrefix*, *diff.dstPrefix*, and *diff.mnemonicPrefix* (see *git-config*(1)).

--line-prefix=<prefix>

Prepend an additional prefix to every line of output.

--ita-invisible-in-index

By default entries added by "git add -N" appear as an existing empty file in "git diff" and a new file in "git diff --cached". This option makes the entry appear as a new file in "git diff" and non-existent in "git diff --cached". This option could be reverted with *--ita-visible-in-index*. Both options are experimental and could be removed in future.

For more detailed explanation on these common options, see also Section G.4.4, “gitdiffcore(7)”.

Generating patch text with -p

Running Section G.3.44, “git-diff(1)”, Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, Section G.3.42, “git-diff-index(1)”, Section G.3.43, “git-diff-tree(1)”, or Section G.3.41, “git-diff-files(1)” with the *-p* option produces patch text. You can customize the creation of patch text via the *GIT_EXTERNAL_DIFF* and the *GIT_DIFF_OPTS* environment variables (see Section G.3.1, “git(1)”), and the *diff* attribute (see Section G.4.2, “gitattributes(5)”).

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded by a "git diff" header that looks like this:

```
diff --git a/file1 b/file2
```

The *a/* and *b/* filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, */dev/null* is *not* used in place of the *a/* or *b/* filenames.

When a rename/copy is involved, *file1* and *file2* show the name of the source file of the rename/copy and the name of the file that the rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

```
old mode <mode>
new mode <mode>
deleted file mode <mode>
new file mode <mode>
copy from <path>
copy to <path>
rename from <path>
rename to <path>
similarity index <number>
dissimilarity index <number>
index <hash>..<hash> <mode>
```

File modes are printed as 6-digit octal numbers including the file type and file permission bits.

Path names in extended headers do not include the *a/* and *b/* prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the blob object names before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. Pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

4. All the *file1* files in the output refer to files before the commit, and all the *file2* files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

5. Hunk headers mention the name of the function to which the hunk applies. See "Defining a custom hunk-header" in Section G.4.2, “gitattributes(5)” for details of how to tailor this to specific languages.

Combined diff format

Any diff-generating command can take the *-c* or *--cc* option to produce a *combined diff* when showing a merge. This is the default format when showing merges with Section G.3.44, “git-diff(1)” or Section G.3.134, “git-show(1)”. Note also that you can give suitable *--diff-merges* option to any of these commands to force generation of diffs in a specific format.

A "combined diff" format looks like this:

```
diff --combined describe.c
```

```
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
        return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +      unsigned char sha1[20];
 +      struct commit *cmit;
        struct commit_list *list;
        static int initialized = 0;
        struct commit_name *n;

 +      if (get_sha1(arg, sha1) < 0)
 +              usage(describe_usage);
 +      cmit = lookup_commit_reference(sha1);
 +      if (!cmit)
 +              usage(describe_usage);
 +
        if (!initialized) {
                initialized = 1;
                for_each_ref(get_name);
```

1. It is preceded by a "git diff" header, that looks like this (when the *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when the *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index <hash>,<hash>..<hash>
   mode <mode>,<mode>..<mode>
   new file mode <mode>
   deleted file mode <mode>,<mode>
   ```

   The *mode <mode>,<mode>..<mode>* line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected content movement (renames and copying detection) are designed to work with the diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by a two-line from-file/to-file header:

   ```
   --- a/file
   +++ b/file
   ```

   Similar to the two-line header for the traditional *unified* diff format, */dev/null* is used to signal created or deleted files.

   However, if the --combined-all-paths option is provided, instead of a two-line from-file/to-file, you get an N +1 line from-file/to-file header, where N is the number of parents in the merge commit:

   ```
   --- a/file
   --- a/file
   --- a/file
   ```

```
+++ b/file
```

This extended format can be useful if rename or copy detection is active, to allow you to see the original name of the file in different parents.

4. Chunk header format is modified to prevent people from accidentally feeding it to *patch -p1*. Combined diff format was created for review of merge commit changes, and was not meant to be applied. The change is similar to the change in the extended *index* header:

```
@@@ <from-file-range> <from-file-range> <to-file-range> @@@
```

There are (number of parents + 1) @ characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has - (minus -- appears in A but removed in B), + (plus -- missing in A but added to B), or " " (space -- unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X's line is different from it.

A - character in the column N means that the line appears in fileN but it does not appear in the result. A + character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two - removals from both file1 and file2, plus ++ to mean one line that was added does not appear in either file1 or file2). Also, eight other lines are the same from file1 but do not appear in file2 (hence prefixed with +).

When shown by *git diff-tree -c*, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by *git diff-files -c*, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

EXAMPLES

git log --no-merges

Show the whole commit history, but skip any merges

git log v2.6.12.. include/scsi drivers/scsi

Show all commits since version *v2.6.12* that changed any file in the *include/scsi* or *drivers/scsi* subdirectories

git log --since="2 weeks ago" -- gitk

Show the changes during the last two weeks to the file *gitk*. The -- is necessary to avoid confusion with the **branch** named *gitk*

git log --name-status release..test

Show the commits that are in the "test" branch but not yet in the "release" branch, along with the list of paths each commit modifies.

git log --follow builtin/rev-list.c

Shows the commits that changed *builtin/rev-list.c*, including those commits that occurred before the file was given its present name.

git log --branches --not --remotes=origin

Shows all commits that are in any of local branches but not in any of remote-tracking branches for *origin* (what you have that origin doesn't).

git log master --not --remotes=/master*

Shows all commits that are in local master but not in any remote repository master branches.

git log -p -m --first-parent

Shows the history including change diffs, but only from the main branch perspective, skipping commits that come from merged branches, and showing full diffs of changes introduced by the merges. This makes sense only when following a strict policy of merging all topic branches when staying on a single integration branch.

git log -L '/int main/',/^}/:main.c

Shows how the function *main()* in the file *main.c* evolved over time.

git log -3

Limits the number of commits to show to 3.

DISCUSSION

Git is to some extent character encoding agnostic.

- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- Path names are encoded in UTF-8 normalization form C. This applies to tree objects, the index file, ref names, as well as path names in command line arguments, environment variables and config files (*.git/config* (see Section G.3.29, "git-config(1)"), Section G.4.5, "gitignore(5)", Section G.4.2, "gitattributes(5)" and Section G.4.10, "gitmodules(5)").

  Note that Git at the core level treats path names simply as sequences of non-NUL bytes, there are no path name encoding conversions (except on Mac and Windows). Therefore, using non-ASCII path names will mostly work even on platforms and file systems that use legacy extended ASCII encodings. However, repositories created on such systems will not work properly on UTF-8-based systems (e.g. Linux, Mac, Windows) and vice versa. Additionally, many Git-based tools simply assume path names to be UTF-8 and will fail to display other encodings correctly.
- Commit log messages are typically encoded in UTF-8, but other extended ASCII encodings are also supported. This includes ISO-8859-x, CP125x and many others, but *not* UTF-16/32, EBCDIC and CJK multi-byte encodings (GBK, Shift-JIS, Big5, EUC-x, CP9xx etc.).

Although we encourage that the commit log messages are encoded in UTF-8, both the core and Git Porcelain are designed not to force UTF-8 on projects. If all participants of a particular project find it more convenient to use legacy encodings, Git does not forbid it. However, there are a few things to keep in mind.

1. *git commit* and *git commit-tree* issue a warning if the commit log message given to it does not look like a valid UTF-8 string, unless you explicitly say your project uses a legacy encoding. The way to say this is to have *i18n.commitEncoding* in *.git/config* file, like this:

   ```
   [i18n]
           commitEncoding = ISO-8859-1
   ```

   Commit objects created with the above setting record the value of *i18n.commitEncoding* in their *encoding* header. This is to help other people who look at them later. Lack of this header implies that the commit log message is encoded in UTF-8.

2. *git log*, *git show*, *git blame* and friends look at the *encoding* header of a commit object, and try to re-code the log message into UTF-8 unless otherwise specified. You can specify the desired output encoding with *i18n.logOutputEncoding* in *.git/config* file, like this:

   ```
   [i18n]
           logOutputEncoding = ISO-8859-1
   ```

   If you do not have this configuration variable, the value of *i18n.commitEncoding* is used instead.

Note that we deliberately chose not to re-code the commit log message when a commit is made to force UTF-8 at the commit object level, because re-coding to UTF-8 is not necessarily a reversible operation.

CONFIGURATION

See Section G.3.29, “git-config(1)” for core variables and Section G.3.44, “git-diff(1)” for settings related to diff generation.

format.pretty

Default for the *--format* option. (See *Pretty Formats* above.) Defaults to *medium*.

i18n.logOutputEncoding

Encoding to use when displaying logs. (See *Discussion* above.) Defaults to the value of *i18n.commitEncoding* if set, and UTF-8 otherwise.

Everything above this line in this section isn't included from the Section G.3.29, “git-config(1)” documentation. The content that follows is the same as what's found there:

log.abbrevCommit

If true, makes Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, and Section G.3.158, “git-whatchanged(1)” assume *--abbrev-commit*. You may override this option with *--no-abbrev-commit*.

log.date

Set the default date-time mode for the *log* command. Setting a value for log.date is similar to using *git log*'s *--date* option. See Section G.3.74, “git-log(1)” for details.

If the format is set to "auto:foo" and the pager is in use, format "foo" will be used for the date format. Otherwise, "default" will be used.

log.decorate

Print out the ref names of any commits that are shown by the log command. If *short* is specified, the ref name prefixes *refs/heads/*, *refs/tags/* and *refs/remotes/* will not be printed. If *full* is specified, the full ref name (including prefix) will be printed. If *auto* is specified, then if the output is going to a terminal, the ref names are shown as if *short* were given, otherwise no ref names are shown. This is the same as the *--decorate* option of the *git log*.

log.initialDecorationSet

By default, *git log* only shows decorations for certain known ref namespaces. If *all* is specified, then show all refs as decorations.

log.excludeDecoration

Exclude the specified patterns from the log decorations. This is similar to the *--decorate-refs-exclude* command-line option, but the config option can be overridden by the *--decorate-refs* option.

log.diffMerges

Set diff format to be used when *--diff-merges=on* is specified, see *--diff-merges* in Section G.3.74, “git-log(1)” for details. Defaults to *separate*.

log.follow

If *true*, *git log* will act as if the *--follow* option was used when a single <path> is given. This has the same limitations as *--follow*, i.e. it cannot be used to follow multiple files and does not work well on non-linear history.

log.graphColors

A list of colors, separated by commas, that can be used to draw history lines in *git log --graph*.

log.showRoot

If true, the initial commit will be shown as a big creation event. This is equivalent to a diff against an empty tree. Tools like Section G.3.74, “git-log(1)” or Section G.3.158, “git-whatchanged(1)”, which normally hide the root commit will now show it. True by default.

log.showSignature

If true, makes Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, and Section G.3.158, “git-whatchanged(1)” assume *--show-signature*.

log.mailmap

If true, makes Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, and Section G.3.158, “git-whatchanged(1)” assume *--use-mailmap*, otherwise assume *--no-use-mailmap*. True by default.

notes.mergeStrategy

Which merge strategy to choose by default when resolving notes conflicts. Must be one of *manual*, *ours*, *theirs*, *union*, or *cat_sort_uniq*. Defaults to *manual*. See the "NOTES MERGE STRATEGIES" section of Section G.3.94, “git-notes(1)” for more information on each strategy.

This setting can be overridden by passing the *--strategy* option to Section G.3.94, “git-notes(1)”.

notes.<name>.mergeStrategy

Which merge strategy to choose when doing a notes merge into refs/notes/<name>. This overrides the more general "notes.mergeStrategy". See the "NOTES MERGE STRATEGIES" section in Section G.3.94, “git-notes(1)” for more information on the available strategies.

notes.displayRef

Which ref (or refs, if a glob or specified more than once), in addition to the default set by *core.notesRef* or *GIT_NOTES_REF*, to read notes from when showing commit messages with the *git log* family of commands.

This setting can be overridden with the *GIT_NOTES_DISPLAY_REF* environment variable, which must be a colon separated list of refs or globs.

A warning will be issued for refs that do not exist, but a glob that does not match any refs is silently ignored.

This setting can be disabled by the *--no-notes* option to the *git log* family of commands, or by the *--notes=<ref>* option accepted by those commands.

The effective value of "core.notesRef" (possibly overridden by GIT_NOTES_REF) is also implicitly added to the list of refs to be displayed.

notes.rewrite.<command>

When rewriting commits with <command> (currently *amend* or *rebase*), if this variable is *false*, git will not copy notes from the original to the rewritten commit. Defaults to *true*. See also "*notes.rewriteRef*" below.

This setting can be overridden with the *GIT_NOTES_REWRITE_REF* environment variable, which must be a colon separated list of refs or globs.

notes.rewriteMode

When copying notes during a rewrite (see the "notes.rewrite.<command>" option), determines what to do if the target commit already has a note. Must be one of *overwrite*, *concatenate*, *cat_sort_uniq*, or *ignore*. Defaults to *concatenate*.

This setting can be overridden with the *GIT_NOTES_REWRITE_MODE* environment variable.

notes.rewriteRef

When copying notes during a rewrite, specifies the (fully qualified) ref whose notes should be copied. May be a glob, in which case notes in all matching refs will be copied. You may also specify this configuration several times.

Does not have a default value; you must configure this variable to enable note rewriting. Set it to *refs/notes/commits* to enable rewriting for the default commit notes.

Can be overridden with the *GIT_NOTES_REWRITE_REF* environment variable. See *notes.rewrite.<command>* above for a further description of its format.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.75. git-ls-files(1)

2024-07-02
Revision History
2024-07-02

NAME

git-ls-files - Show information about files in the index and the working tree

SYNOPSIS

git ls-files [-z] [-t] [-v] [-f]
[-c|--cached] [-d|--deleted] [-o|--others] [-i|--ignored]
[-s|--stage] [-u|--unmerged] [-k|--killed] [-m|--modified]
[--resolve-undo]
[--directory [--no-empty-directory]] [--eol]
[--deduplicate]
[-x <pattern>|--exclude=<pattern>]
[-X <file>|--exclude-from=<file>]
[--exclude-per-directory=<file>]
[--exclude-standard]
[--error-unmatch] [--with-tree=<tree-ish>]
[--full-name] [--recurse-submodules]
[--abbrev[=<n>]] [--format=<format>] [--] [<file>…]

DESCRIPTION

This command merges the file listing in the index with the actual working directory list, and shows different combinations of the two.

Several flags can be used to determine which files are shown, and each file may be printed multiple times if there are multiple entries in the index or if multiple statuses are applicable for the relevant file selection options.

OPTIONS

-c , --cached

Show all files cached in Git's index, i.e. all tracked files. (This is the default if no -c/-s/-d/-o/-u/-k/-m/--resolve-undo options are specified.)

-d , --deleted

Show files with an unstaged deletion

-m , --modified

Show files with an unstaged modification (note that an unstaged deletion also counts as an unstaged modification)

-o , --others

Show other (i.e. untracked) files in the output

-i , --ignored

Show only ignored files in the output. Must be used with either an explicit *-c* or *-o*. When showing files in the index (i.e. when used with *-c*), print only those files matching an exclude pattern. When showing "other" files (i.e. when used with *-o*), show only those matched by an exclude pattern. Standard ignore rules are not automatically activated; therefore, at least one of the *--exclude** options is required.

-s , --stage

Show staged contents' mode bits, object name and stage number in the output.

--directory

If a whole directory is classified as "other", show just its name (with a trailing slash) and not its whole contents. Has no effect without -o/--others.

--no-empty-directory

Do not list empty directories. Has no effect without --directory.

-u , --unmerged

Show information about unmerged files in the output, but do not show any other tracked files (forces --stage, overrides --cached).

-k , --killed

Show untracked files on the filesystem that need to be removed due to file/directory conflicts for tracked files to be able to be written to the filesystem.

--resolve-undo

Show files having resolve-undo information in the index together with their resolve-undo information. (resolve-undo information is what is used to implement "git checkout -m $PATH", i.e. to recreate merge conflicts that were accidentally resolved)

-z

\0 line termination on output and do not quote filenames. See OUTPUT below for more information.

--deduplicate

When only filenames are shown, suppress duplicates that may come from having multiple stages during a merge, or giving *--deleted* and *--modified* option at the same time. When any of the *-t*, *--unmerged*, or *--stage* option is in use, this option has no effect.

-x <pattern> , --exclude=<pattern>

Skip untracked files matching pattern. Note that pattern is a shell wildcard pattern. See EXCLUDE PATTERNS below for more information.

-X <file> , --exclude-from=<file>

Read exclude patterns from <file>; 1 per line.

--exclude-per-directory=<file>

Read additional exclude patterns that apply only to the directory and its subdirectories in <file>. If you are trying to emulate the way Porcelain commands work, using the *--exclude-standard* option instead is easier and more thorough.

--exclude-standard

Add the standard Git exclusions: .git/info/exclude, .gitignore in each directory, and the user's global exclusion file.

--error-unmatch

If any <file> does not appear in the index, treat this as an error (return 1).

--with-tree=<tree-ish>

When using --error-unmatch to expand the user supplied <file> (i.e. path pattern) arguments to paths, pretend that paths which were removed in the index since the named <tree-ish> are still present. Using this option with *-s* or *-u* options does not make any sense.

-t

Show status tags together with filenames. Note that for scripting purposes, Section G.3.138, “git-status(1)” *--porcelain* and Section G.3.41, “git-diff-files(1)” *--name-status* are almost always superior alternatives; users should look at Section G.3.138, “git-status(1)” *--short* or Section G.3.44, “git-diff(1)” *--name-status* for more user-friendly alternatives.

This option provides a reason for showing each filename, in the form of a status tag (which is followed by a space and then the filename). The status tags are all single characters from the following list:

H

tracked file that is not either unmerged or skip-worktree

S

tracked file that is skip-worktree

M

tracked file that is unmerged

R

tracked file with unstaged removal/deletion

C

tracked file with unstaged modification/change

K

untracked paths which are part of file/directory conflicts which prevent checking out tracked files

?

untracked file

U

file with resolve-undo information

-v

Similar to *-t*, but use lowercase letters for files that are marked as *assume unchanged* (see Section G.3.147, “git-update-index(1)”).

-f

Similar to *-t*, but use lowercase letters for files that are marked as *fsmonitor valid* (see Section G.3.147, “git-update-index(1)”).

--full-name

When run from a subdirectory, the command usually outputs paths relative to the current directory. This option forces paths to be output relative to the project top directory.

--recurse-submodules

Recursively calls ls-files on each active submodule in the repository. Currently there is only support for the --cached and --stage modes.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object lines, show the shortest prefix that is at least <*n*> hexdigits long that uniquely refers the object. Non default number of digits can be specified with --abbrev=<n>.

--debug

After each line that describes a file, add more data about its cache entry. This is intended to show as much information as possible for manual inspection; the exact format may change at any time.

--eol

Show <eolinfo> and <eolattr> of files. <eolinfo> is the file content identification used by Git when the "text" attribute is "auto" (or not set and core.autocrlf is not false). <eolinfo> is either "-text", "none", "lf", "crlf", "mixed" or "".

"" means the file is not a regular file, it is not in the index or not accessible in the working tree.

<eolattr> is the attribute that is used when checking out or committing, it is either "", "-text", "text", "text=auto", "text eol=lf", "text eol=crlf". Since Git 2.10 "text=auto eol=lf" and "text=auto eol=crlf" are supported.

Both the <eolinfo> in the index ("i/<eolinfo>") and in the working tree ("w/<eolinfo>") are shown for regular files, followed by the ("attr/<eolattr>").

--sparse

If the index is sparse, show the sparse directories without expanding to the contained files. Sparse directories will be shown with a trailing slash, such as "x/" for a sparse directory "x".

--format=<format>

A string that interpolates *%(fieldname)* from the result being shown. It also interpolates *%%* to *%*, and *%xx* where *xx* are hex digits interpolates to character with hex code *xx*; for example *%00* interpolates to *\0* (NUL), *%09* to *\t* (TAB) and %0a to *\n* (LF). --format cannot be combined with *-s*, *-o*, *-k*, *-t*, *--resolve-undo* and *--eol*.

--

Do not interpret any more arguments as options.

<file>

Files to show. If no files are given all files which match the other specified criteria are shown.

OUTPUT

git ls-files just outputs the filenames unless *--stage* is specified in which case it outputs:

```
[<tag> ]<mode> <object> <stage> <file>
```

git ls-files --eol will show i/<eolinfo><SPACES>w/<eolinfo><SPACES>attr/<eolattr><SPACE*><TAB><file>

git ls-files --unmerged and *git ls-files --stage* can be used to examine detailed information on unmerged paths.

For an unmerged path, instead of recording a single mode/SHA-1 pair, the index records up to three such pairs; one from tree O in stage 1, A in stage 2, and B in stage 3. This information can be used by the user (or the porcelain) to see what should eventually be recorded at the path. (see Section G.3.106, “git-read-tree(1)” for more information on state)

Without the *-z* option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). Using *-z* the filename is output verbatim and the line is terminated by a NUL byte.

It is possible to print in a custom format by using the *--format* option, which is able to interpolate different fields using a *%(fieldname)* notation. For example, if you only care about the "objectname" and "path" fields, you can execute with a specific "--format" like

```
git ls-files --format='%(objectname) %(path)'
```

FIELD NAMES

The way each path is shown can be customized by using the *--format=<format>* option, where the %(fieldname) in the <format> string for various aspects of the index entry are interpolated. The following "fieldname" are understood:

objectmode

The mode of the file which is recorded in the index.

objecttype

The object type of the file which is recorded in the index.

objectname

The name of the file which is recorded in the index.

objectsize[:padded]

The object size of the file which is recorded in the index ("-" if the object is a *commit* or *tree*). It also supports a padded format of size with "%(objectsize:padded)".

stage

The stage of the file which is recorded in the index.

eolinfo:index , eolinfo:worktree

The <eolinfo> (see the description of the *--eol* option) of the contents in the index or in the worktree for the path.

eolattr

The <eolattr> (see the description of the *--eol* option) that applies to the path.

path

The pathname of the file which is recorded in the index.

EXCLUDE PATTERNS

git ls-files can use a list of "exclude patterns" when traversing the directory tree and finding files to show when the flags --others or --ignored are specified. Section G.4.5, “gitignore(5)” specifies the format of exclude patterns.

These exclude patterns can be specified from the following places, in order:

1. The command-line flag --exclude=<pattern> specifies a single pattern. Patterns are ordered in the same order they appear in the command line.
2. The command-line flag --exclude-from=<file> specifies a file containing a list of patterns. Patterns are ordered in the same order they appear in the file.
3. The command-line flag --exclude-per-directory=<name> specifies a name of the file in each directory *git ls-files* examines, normally *.gitignore*. Files in deeper directories take precedence. Patterns are ordered in the same order they appear in the files.

A pattern specified on the command line with --exclude or read from the file specified with --exclude-from is relative to the top of the directory tree. A pattern read from a file specified by --exclude-per-directory is relative to the directory that the pattern file appears in.

Generally, you should be able to use *--exclude-standard* when you want the exclude rules applied the same way as what Porcelain commands do. To emulate what *--exclude-standard* specifies, you can give *--exclude-per-directory=.gitignore*, and then specify:

1. The file specified by the *core.excludesfile* configuration variable, if exists, or the *$XDG_CONFIG_HOME/git/ignore* file.
2. The *$GIT_DIR/info/exclude* file.

via the *--exclude-from=* option.

SEE ALSO

Section G.3.106, “git-read-tree(1)”, Section G.4.5, “gitignore(5)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.76. git-ls-remote(1)

2024-07-02

Revision History

2024-07-02

NAME

git-ls-remote - List references in a remote repository

SYNOPSIS

git ls-remote [--heads] [--tags] [--refs] [--upload-pack=<exec>]
[-q | --quiet] [--exit-code] [--get-url] [--sort=<key>]
[--symref] [<repository> [<patterns>…]]

DESCRIPTION

Displays references available in a remote repository along with the associated commit IDs.

OPTIONS

-h , --heads , -t , --tags

Limit to only refs/heads and refs/tags, respectively. These options are *not* mutually exclusive; when given both, references stored in refs/heads and refs/tags are displayed. Note that *git ls-remote -h* used without anything else on the command line gives help, consistent with other git subcommands.

--refs

Do not show peeled tags or pseudorefs like *HEAD* in the output.

-q , --quiet

Do not print remote URL to stderr.

--upload-pack=<exec>

Specify the full path of *git-upload-pack* on the remote host. This allows listing references from repositories accessed via SSH and where the SSH daemon does not use the PATH configured by the user.

--exit-code

Exit with status "2" when no matching refs are found in the remote repository. Usually the command exits with status "0" to indicate it successfully talked with the remote repository, whether it found any matching refs.

--get-url

Expand the URL of the given remote repository taking into account any "url.<base>.insteadOf" config setting (See Section G.3.29, “git-config(1)”) and exit without talking to the remote.

--symref

In addition to the object pointed by it, show the underlying ref pointed by it when showing a symbolic ref. Currently, upload-pack only shows the symref HEAD, so it will be the only one shown by ls-remote.

--sort=<key>

Sort based on the key given. Prefix - to sort in descending order of the value. Supports "version:refname" or "v:refname" (tag names are treated as versions). The "version:refname" sort order can also be affected by the "versionsort.suffix" configuration variable. See Section G.3.52, “git-for-each-ref(1)” for more sort options, but be aware keys like *committerdate* that require access to the objects themselves will not work for refs whose objects have not yet been fetched from the remote, and will give a *missing object* error.

-o <option> , --server-option=<option>

Transmit the given string to the server when communicating using protocol version 2. The given string must not contain a NUL or LF character. When multiple *--server-option=<option>* are given, they are all sent to the other side in the order listed on the command line.

<repository>

The "remote" repository to query. This parameter can be either a URL or the name of a remote (see the GIT URLS and REMOTES sections of Section G.3.49, “git-fetch(1)”).

<patterns>…

When unspecified, all references, after filtering done with --heads and --tags, are shown. When <patterns>… are specified, only references matching one or more of the given patterns are displayed. Each pattern is interpreted as a glob (see *glob* in Section G.4.19, “gitglossary(7)”) which is matched against the "tail" of a ref,

starting either from the start of the ref (so a full name like *refs/heads/foo* matches) or from a slash separator (so *bar* matches *refs/heads/bar* but not *refs/heads/foobar*).

OUTPUT

The output is in the format:

```
<oid> TAB <ref> LF
```

When showing an annotated tag, unless *--refs* is given, two such lines are shown: one with the refname for the tag itself as *<ref>*, and another with *<ref>* followed by *^{}*. The *<oid>* on the latter line shows the name of the object the tag points at.

EXAMPLES

- List all references (including symbolics and pseudorefs), peeling tags:

```
$ git ls-remote
27d43aaaf50ef0ae014b88bba294f93658016a2e        HEAD
950264636c68591989456e3ba0a5442f93152c1a        refs/heads/main
d9ab777d41f92a8c1684c91cfb02053d7dd1046b        refs/heads/next
d4ca2e3147b409459955613c152220f4db848ee1        refs/tags/v2.40.0
73876f4861cd3d187a4682290ab75c9dccadbc56        refs/tags/v2.40.0^{}
```

- List all references matching given patterns:

```
$ git ls-remote http://www.kernel.org/pub/scm/git/git.git master seen rc
5fe978a5381f1fbad26a80e682ddd2a401966740        refs/heads/master
c781a84b5204fb294c9ccc79f8b3baceeb32c061        refs/heads/seen
```

- List only tags matching a given wildcard pattern:

```
$ git ls-remote --tags http://www.kernel.org/pub/scm/git/git.git v\*
485a869c64a68cc5795dd99689797c5900f4716d        refs/tags/v2.39.2
cbf04937d5b9fcf0a76c28f69e6294e9e3ecd7e6        refs/tags/v2.39.2^{}
d4ca2e3147b409459955613c152220f4db848ee1        refs/tags/v2.40.0
73876f4861cd3d187a4682290ab75c9dccadbc56        refs/tags/v2.40.0^{}
```

SEE ALSO

Section G.3.17, “git-check-ref-format(1)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.77. git-ls-tree(1)

2024-07-02

Revision History

2024-07-02

NAME

git-ls-tree - List the contents of a tree object

SYNOPSIS

git ls-tree [-d] [-r] [-t] [-l] [-z]
[--name-only] [--name-status] [--object-only] [--full-name] [--full-tree] [--abbrev[=<n>]] [--format=<format>]
<tree-ish> [<path>…]

DESCRIPTION

Lists the contents of a given tree object, like what "/bin/ls -a" does in the current working directory. Note that:

- the behaviour is slightly different from that of "/bin/ls" in that the <*path*> denotes just a list of patterns to match, e.g. so specifying directory name (without *-r*) will behave differently, and order of the arguments does not matter.
- the behaviour is similar to that of "/bin/ls" in that the <*path*> is taken as relative to the current working directory. E.g. when you are in a directory *sub* that has a directory *dir*, you can run *git ls-tree -r HEAD dir* to list the contents of the tree (that is *sub/dir* in *HEAD*). You don't want to give a tree that is not at the root level (e.g. *git ls-tree -r HEAD:sub dir*) in this case, as that would result in asking for *sub/sub/dir* in the *HEAD* commit. However, the current working directory can be ignored by passing --full-tree option.

OPTIONS

<tree-ish>

Id of a tree-ish.

-d

Show only the named tree entry itself, not its children.

-r

Recurse into sub-trees.

-t

Show tree entries even when going to recurse them. Has no effect if *-r* was not passed. *-d* implies *-t*.

-l , --long

Show object size of blob (file) entries.

-z

\0 line termination on output and do not quote filenames. See OUTPUT FORMAT below for more information.

--name-only , --name-status

List only filenames (instead of the "long" output), one per line. Cannot be combined with *--object-only*.

--object-only

List only names of the objects, one per line. Cannot be combined with *--name-only* or *--name-status*. This is equivalent to specifying *--format='%(objectname)'*, but for both this option and that exact format the command takes a hand-optimized codepath instead of going through the generic formatting mechanism.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object lines, show the shortest prefix that is at least <*n*> hexdigits long that uniquely refers the object. Non default number of digits can be specified with --abbrev=<n>.

--full-name

Instead of showing the path names relative to the current working directory, show the full path names.

--full-tree

Do not limit the listing to the current working directory. Implies --full-name.

--format=<format>

A string that interpolates *%(fieldname)* from the result being shown. It also interpolates *%%* to *%*, and *%xNN* where *NN* are hex digits interpolates to character with hex code *NN*; for example *%x00* interpolates to *\0* (NUL), *%x09* to *\t* (TAB) and *%x0a* to *\n* (LF). When specified, *--format* cannot be combined with other format-altering options, including *--long*, *--name-only* and *--object-only*.

[<path>…]

When paths are given, show them (note that this isn't really raw pathnames, but rather a list of patterns to match). Otherwise implicitly uses the root level of the tree as the sole path argument.

Output Format

The output format of *ls-tree* is determined by either the *--format* option, or other format-altering options such as *--name-only* etc. (see *--format* above).

The use of certain *--format* directives is equivalent to using those options, but invoking the full formatting machinery can be slower than using an appropriate formatting option.

In cases where the *--format* would exactly map to an existing option *ls-tree* will use the appropriate faster path. Thus the default format is equivalent to:

```
%(objectmode) %(objecttype) %(objectname)%x09%(path)
```

This output format is compatible with what *--index-info --stdin* of *git update-index* expects.

When the *-l* option is used, format changes to

```
%(objectmode) %(objecttype) %(objectname) %(objectsize:padded)%x09%(path)
```

Object size identified by <objectname> is given in bytes, and right-justified with minimum width of 7 characters. Object size is given only for blobs (file) entries; for other entries - character is used in place of size.

Without the *-z* option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). Using *-z* the filename is output verbatim and the line is terminated by a NUL byte.

Customized format:

It is possible to print in a custom format by using the *--format* option, which is able to interpolate different fields using a *%(fieldname)* notation. For example, if you only care about the "objectname" and "path" fields, you can execute with a specific "--format" like

```
git ls-tree --format='%(objectname) %(path)' <tree-ish>
```

FIELD NAMES

Various values from structured fields can be used to interpolate into the resulting output. For each outputting line, the following names can be used:

objectmode

The mode of the object.

objecttype

The type of the object (*commit*, *blob* or *tree*).

objectname

The name of the object.

objectsize[:padded]

The size of a *blob* object ("-" if it's a *commit* or *tree*). It also supports a padded format of size with "%(objectsize:padded)".

path

The pathname of the object.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.78. git-mailinfo(1)

2024-07-02
Revision History
2024-07-02

NAME

git-mailinfo - Extracts patch and authorship from a single e-mail message

SYNOPSIS

git mailinfo [-k|-b] [-u | --encoding=<encoding> | -n]
[--[no-]scissors] [--quoted-cr=<action>]
<msg> <patch>

DESCRIPTION

Reads a single e-mail message from the standard input, and writes the commit log message in <msg> file, and the patches in <patch> file. The author name, e-mail and e-mail subject are written out to the standard output to be used by *git am* to create a commit. It is usually not necessary to use this command directly. See Section G.3.3, “git-am(1)” instead.

OPTIONS

-k

Usually the program removes email cruft from the Subject: header line to extract the title line for the commit log message. This option prevents this munging, and is most useful when used to read back *git format-patch -k* output.

Specifically, the following are removed until none of them remain:

- Leading and trailing whitespace.
- Leading *Re:*, *re:*, and *:*.
- Leading bracketed strings (between *[* and *]*, usually *[PATCH]*).

Finally, runs of whitespace are normalized to a single ASCII space character.

-b

When -k is not in effect, all leading strings bracketed with *[* and *]* pairs are stripped. This option limits the stripping to only the pairs whose bracketed string contains the word "PATCH".

-u

The commit log message, author name and author email are taken from the e-mail, and after minimally decoding MIME transfer encoding, re-coded in the charset specified by *i18n.commitEncoding* (defaulting to UTF-8) by transliterating them. This used to be optional but now it is the default.

Note that the patch is always used as-is without charset conversion, even with this flag.

--encoding=<encoding>

Similar to -u. But when re-coding, the charset specified here is used instead of the one specified by *i18n.commitEncoding* or UTF-8.

-n

Disable all charset re-coding of the metadata.

-m , --message-id

Copy the Message-ID header at the end of the commit message. This is useful in order to associate commits with mailing list discussions.

--scissors

Remove everything in body before a scissors line (e.g. "-- >8 --"). The line represents scissors and perforation marks, and is used to request the reader to cut the message at that line. If that line appears in the body of the message before the patch, everything before it (including the scissors line itself) is ignored when this option is used.

This is useful if you want to begin your message in a discussion thread with comments and suggestions on the message you are responding to, and to conclude it with a patch submission, separating the discussion and the beginning of the proposed commit log message with a scissors line.

This can be enabled by default with the configuration option mailinfo.scissors.

--no-scissors

Ignore scissors lines. Useful for overriding mailinfo.scissors settings.

--quoted-cr=<action>

Action when processes email messages sent with base64 or quoted-printable encoding, and the decoded lines end with a CRLF instead of a simple LF.

The valid actions are:

- *nowarn*: Git will do nothing when such a CRLF is found.
- *warn*: Git will issue a warning for each message if such a CRLF is found.
- *strip*: Git will convert those CRLF to LF.

The default action could be set by configuration option *mailinfo.quotedCR*. If no such configuration option has been set, *warn* will be used.

<msg>

The commit log message extracted from e-mail, usually except the title line which comes from e-mail Subject.

<patch>

The patch extracted from e-mail.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

mailinfo.scissors

If true, makes Section G.3.78, “git-mailinfo(1)” (and therefore Section G.3.3, “git-am(1)”) act by default as if the --scissors option was provided on the command-line. When active, this feature removes everything from the message body before a scissors line (i.e. consisting mainly of ">8", "8<" and "-").

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.79. git-mailsplit(1)

2024-07-02

Revision History

2024-07-02

NAME

git-mailsplit - Simple UNIX mbox splitter program

SYNOPSIS

git mailsplit [-b] [-f<nn>] [-d<prec>] [--keep-cr] [--mboxrd]
-o<directory> [--] [(<mbox>|<Maildir>)…]

DESCRIPTION

Splits a mbox file or a Maildir into a list of files: "0001" "0002" .. in the specified directory so you can process them further from there.

> **Important**
>
> Maildir splitting relies upon filenames being sorted to output patches in the correct order.

OPTIONS

<mbox>

Mbox file to split. If not given, the mbox is read from the standard input.

<Maildir>

Root of the Maildir to split. This directory should contain the cur, tmp and new subdirectories.

-o<directory>

Directory in which to place the individual messages.

-b

If any file doesn't begin with a From line, assume it is a single mail message instead of signaling an error.

-d<prec>

Instead of the default 4 digits with leading zeros, different precision can be specified for the generated filenames.

-f<nn>

Skip the first <nn> numbers, for example if -f3 is specified, start the numbering with 0004.

--keep-cr

Do not remove *\r* from lines ending with *\r\n*.

--mboxrd

Input is of the "mboxrd" format and "^>+From " line escaping is reversed.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.80. git-maintenance(1)

2024-07-02
Revision History
2024-07-02

NAME

git-maintenance - Run tasks to optimize Git repository data

SYNOPSIS

git maintenance run [<options>]
git maintenance start [--scheduler=<scheduler>]
git maintenance (stop|register|unregister) [<options>]

DESCRIPTION

Run tasks to optimize Git repository data, speeding up other Git commands and reducing storage requirements for the repository.

Git commands that add repository data, such as *git add* or *git fetch*, are optimized for a responsive user experience. These commands do not take time to optimize the Git data, since such optimizations scale with the full size of the repository while these user commands each perform a relatively small action.

The *git maintenance* command provides flexibility for how to optimize the Git repository.

SUBCOMMANDS

run

Run one or more maintenance tasks. If one or more *--task* options are specified, then those tasks are run in that order. Otherwise, the tasks are determined by which *maintenance.<task>.enabled* config options are true. By default, only *maintenance.gc.enabled* is true.

start

Start running maintenance on the current repository. This performs the same config updates as the *register* subcommand, then updates the background scheduler to run *git maintenance run --scheduled* on an hourly basis.

stop

Halt the background maintenance schedule. The current repository is not removed from the list of maintained repositories, in case the background maintenance is restarted later.

register

Initialize Git config values so any scheduled maintenance will start running on this repository. This adds the repository to the *maintenance.repo* config variable in the current user's global config, or the config specified by --config-file option, and enables some recommended configuration values for *maintenance.<task>.schedule*. The tasks that are enabled are safe for running in the background without disrupting foreground processes.

The *register* subcommand will also set the *maintenance.strategy* config value to *incremental*, if this value is not previously set. The *incremental* strategy uses the following schedule for each maintenance task:

- *gc*: disabled.
- *commit-graph*: hourly.
- *prefetch*: hourly.
- *loose-objects*: daily.
- *incremental-repack*: daily.

git maintenance register will also disable foreground maintenance by setting *maintenance.auto = false* in the current repository. This config setting will remain after a *git maintenance unregister* command.

unregister

Remove the current repository from background maintenance. This only removes the repository from the configured list. It does not stop the background maintenance processes from running.

The *unregister* subcommand will report an error if the current repository is not already registered. Use the *--force* option to return success even when the current repository is not registered.

TASKS

commit-graph

The *commit-graph* job updates the *commit-graph* files incrementally, then verifies that the written data is correct. The incremental write is safe to run alongside concurrent Git processes since it will not expire *.graph* files that were in the previous *commit-graph-chain* file. They will be deleted by a later run based on the expiration delay.

prefetch

The *prefetch* task updates the object directory with the latest objects from all registered remotes. For each remote, a *git fetch* command is run. The configured refspec is modified to place all requested refs within *refs/prefetch/*. Also, tags are not updated.

This is done to avoid disrupting the remote-tracking branches. The end users expect these refs to stay unmoved unless they initiate a fetch. However, with the prefetch task, the objects necessary to complete a later real fetch would already be obtained, making the real fetch faster. In the ideal case, it will just become an update to a bunch of remote-tracking branches without any object transfer.

gc

Clean up unnecessary files and optimize the local repository. "GC" stands for "garbage collection," but this task performs many smaller tasks. This task can be expensive for large repositories, as it repacks all Git objects into a single pack-file. It can also be disruptive in some situations, as it deletes stale data. See Section G.3.58, “git-gc(1)” for more details on garbage collection in Git.

loose-objects

The *loose-objects* job cleans up loose objects and places them into pack-files. In order to prevent race conditions with concurrent Git commands, it follows a two-step process. First, it deletes any loose objects that already exist in a pack-file; concurrent Git processes will examine the pack-file for the object data instead of the loose object. Second, it creates a new pack-file (starting with "loose-") containing a batch of loose objects. The batch size is limited to 50 thousand objects to prevent the job from taking too long on a repository with many loose objects. The *gc* task writes unreachable objects as loose objects to be cleaned up by a later step only if they are not re-added to a pack-file; for this reason it is not advisable to enable both the *loose-objects* and *gc* tasks at the same time.

incremental-repack

The *incremental-repack* job repacks the object directory using the *multi-pack-index* feature. In order to prevent race conditions with concurrent Git commands, it follows a two-step process. First, it calls *git multi-pack-index*

expire to delete pack-files unreferenced by the *multi-pack-index* file. Second, it calls *git multi-pack-index repack* to select several small pack-files and repack them into a bigger one, and then update the *multi-pack-index* entries that refer to the small pack-files to refer to the new pack-file. This prepares those small pack-files for deletion upon the next run of *git multi-pack-index expire*. The selection of the small pack-files is such that the expected size of the big pack-file is at least the batch size; see the *--batch-size* option for the *repack* subcommand in Section G.3.92, “git-multi-pack-index(1)”. The default batch-size is zero, which is a special case that attempts to repack all pack-files into a single pack-file.

pack-refs

The *pack-refs* task collects the loose reference files and collects them into a single file. This speeds up operations that need to iterate across many references. See Section G.3.98, “git-pack-refs(1)” for more information.

OPTIONS

--auto

When combined with the *run* subcommand, run maintenance tasks only if certain thresholds are met. For example, the *gc* task runs when the number of loose objects exceeds the number stored in the *gc.auto* config setting, or when the number of pack-files exceeds the *gc.autoPackLimit* config setting. Not compatible with the *--schedule* option.

--schedule

When combined with the *run* subcommand, run maintenance tasks only if certain time conditions are met, as specified by the *maintenance.<task>.schedule* config value for each *<task>*. This config value specifies a number of seconds since the last time that task ran, according to the *maintenance.<task>.lastRun* config value. The tasks that are tested are those provided by the *--task=<task>* option(s) or those with *maintenance.<task>.enabled* set to true.

--quiet

Do not report progress or other information over *stderr*.

--task=<task>

If this option is specified one or more times, then only run the specified tasks in the specified order. If no *--task=<task>* arguments are specified, then only the tasks with *maintenance.<task>.enabled* configured as *true* are considered. See the *TASKS* section for the list of accepted *<task>* values.

--scheduler=auto|crontab|systemd-timer|launchctl|schtasks

When combined with the *start* subcommand, specify the scheduler for running the hourly, daily and weekly executions of *git maintenance run*. Possible values for *<scheduler>* are *auto*, *crontab* (POSIX), *systemd-timer* (Linux), *launchctl* (macOS), and *schtasks* (Windows). When *auto* is specified, the appropriate platform-specific scheduler is used; on Linux, *systemd-timer* is used if available, otherwise *crontab*. Default is *auto*.

TROUBLESHOOTING

The *git maintenance* command is designed to simplify the repository maintenance patterns while minimizing user wait time during Git commands. A variety of configuration options are available to allow customizing this process. The default maintenance options focus on operations that complete quickly, even on large repositories.

Users may find some cases where scheduled maintenance tasks do not run as frequently as intended. Each *git maintenance run* command takes a lock on the repository's object database, and this prevents other concurrent *git maintenance run* commands from running on the same repository. Without this safeguard, competing processes could leave the repository in an unpredictable state.

The background maintenance schedule runs *git maintenance run* processes on an hourly basis. Each run executes the "hourly" tasks. At midnight, that process also executes the "daily" tasks. At midnight on the first day of the week, that process also executes the "weekly" tasks. A single process iterates over each registered repository, performing the scheduled tasks for that frequency. Depending on the number of registered repositories and their

sizes, this process may take longer than an hour. In this case, multiple *git maintenance run* commands may run on the same repository at the same time, colliding on the object database lock. This results in one of the two tasks not running.

If you find that some maintenance windows are taking longer than one hour to complete, then consider reducing the complexity of your maintenance tasks. For example, the *gc* task is much slower than the *incremental-repack* task. However, this comes at a cost of a slightly larger object database. Consider moving more expensive tasks to be run less frequently.

Expert users may consider scheduling their own maintenance tasks using a different schedule than is available through *git maintenance start* and Git configuration options. These users should be aware of the object database lock and how concurrent *git maintenance run* commands behave. Further, the *git gc* command should not be combined with *git maintenance run* commands. *git gc* modifies the object database but does not take the lock in the same way as *git maintenance run*. If possible, use *git maintenance run --task=gc* instead of *git gc*.

The following sections describe the mechanisms put in place to run background maintenance by *git maintenance start* and how to customize them.

BACKGROUND MAINTENANCE ON POSIX SYSTEMS

The standard mechanism for scheduling background tasks on POSIX systems is cron(8). This tool executes commands based on a given schedule. The current list of user-scheduled tasks can be found by running *crontab -l*. The schedule written by *git maintenance start* is similar to this:

```
# BEGIN GIT MAINTENANCE SCHEDULE
# The following schedule was created by Git
# Any edits made in this region might be
# replaced in the future by a Git command.

0 1-23 * * * "/<path>/git" --exec-path="/<path>" for-each-repo --
config=maintenance.repo maintenance run --schedule=hourly
0 0 * * 1-6 "/<path>/git" --exec-path="/<path>" for-each-repo --
config=maintenance.repo maintenance run --schedule=daily
0 0 * * 0 "/<path>/git" --exec-path="/<path>" for-each-repo --
config=maintenance.repo maintenance run --schedule=weekly

# END GIT MAINTENANCE SCHEDULE
```

The comments are used as a region to mark the schedule as written by Git. Any modifications within this region will be completely deleted by *git maintenance stop* or overwritten by *git maintenance start*.

The *crontab* entry specifies the full path of the *git* executable to ensure that the executed *git* command is the same one with which *git maintenance start* was issued independent of *PATH*. If the same user runs *git maintenance start* with multiple Git executables, then only the latest executable is used.

These commands use *git for-each-repo --config=maintenance.repo* to run *git maintenance run --schedule=<frequency>* on each repository listed in the multi-valued *maintenance.repo* config option. These are typically loaded from the user-specific global config. The *git maintenance* process then determines which maintenance tasks are configured to run on each repository with each *<frequency>* using the *maintenance.<task>.schedule* config options. These values are loaded from the global or repository config values.

If the config values are insufficient to achieve your desired background maintenance schedule, then you can create your own schedule. If you run *crontab -e*, then an editor will load with your user-specific *cron* schedule. In that editor, you can add your own schedule lines. You could start by adapting the default schedule listed earlier, or you could read the crontab(5) documentation for advanced scheduling techniques. Please do use the full path and *--exec-path* techniques from the default schedule to ensure you are executing the correct binaries in your schedule.

BACKGROUND MAINTENANCE ON LINUX SYSTEMD SYSTEMS

While Linux supports *cron*, depending on the distribution, *cron* may be an optional package not necessarily installed. On modern Linux distributions, systemd timers are superseding it.

If user systemd timers are available, they will be used as a replacement of *cron*.

In this case, *git maintenance start* will create user systemd timer units and start the timers. The current list of user-scheduled tasks can be found by running *systemctl --user list-timers*. The timers written by *git maintenance start* are similar to this:

```
$ systemctl --user list-timers
NEXT                         LEFT          LAST
 PASSED     UNIT                         ACTIVATES
Thu 2021-04-29 19:00:00 CEST 42min left    Thu 2021-04-29 18:00:11 CEST
 17min ago  git-maintenance@hourly.timer git-maintenance@hourly.service
Fri 2021-04-30 00:00:00 CEST 5h 42min left Thu 2021-04-29 00:00:11 CEST 18h
 ago    git-maintenance@daily.timer  git-maintenance@daily.service
Mon 2021-05-03 00:00:00 CEST 3 days left   Mon 2021-04-26 00:00:11 CEST 3
 days ago git-maintenance@weekly.timer git-maintenance@weekly.service
```

One timer is registered for each *--schedule=<frequency>* option.

The definition of the systemd units can be inspected in the following files:

```
~/.config/systemd/user/git-maintenance@.timer
~/.config/systemd/user/git-maintenance@.service
~/.config/systemd/user/timers.target.wants/git-maintenance@hourly.timer
~/.config/systemd/user/timers.target.wants/git-maintenance@daily.timer
~/.config/systemd/user/timers.target.wants/git-maintenance@weekly.timer
```

git maintenance start will overwrite these files and start the timer again with *systemctl --user*, so any customization should be done by creating a drop-in file, i.e. a *.conf* suffixed file in the *~/.config/systemd/user/git-maintenance@.service.d* directory.

git maintenance stop will stop the user systemd timers and delete the above mentioned files.

For more details, see *systemd.timer(5)*.

BACKGROUND MAINTENANCE ON MACOS SYSTEMS

While macOS technically supports *cron*, using *crontab -e* requires elevated privileges and the executed process does not have a full user context. Without a full user context, Git and its credential helpers cannot access stored credentials, so some maintenance tasks are not functional.

Instead, *git maintenance start* interacts with the *launchctl* tool, which is the recommended way to schedule timed jobs in macOS. Scheduling maintenance through *git maintenance (start|stop)* requires some *launchctl* features available only in macOS 10.11 or later.

Your user-specific scheduled tasks are stored as XML-formatted *.plist* files in *~/Library/LaunchAgents/*. You can see the currently-registered tasks using the following command:

```
$ ls ~/Library/LaunchAgents/org.git-scm.git*
org.git-scm.git.daily.plist
org.git-scm.git.hourly.plist
org.git-scm.git.weekly.plist
```

One task is registered for each *--schedule=<frequency>* option. To inspect how the XML format describes each schedule, open one of these *.plist* files in an editor and inspect the *<array>* element following the *<key>StartCalendarInterval</key>* element.

git maintenance start will overwrite these files and register the tasks again with *launchctl*, so any customizations should be done by creating your own *.plist* files with distinct names. Similarly, the *git maintenance stop* command will unregister the tasks with *launchctl* and delete the *.plist* files.

To create more advanced customizations to your background tasks, see launchctl.plist(5) for more information.

BACKGROUND MAINTENANCE ON WINDOWS SYSTEMS

Windows does not support *cron* and instead has its own system for scheduling background tasks. The *git maintenance start* command uses the *schtasks* command to submit tasks to this system. You can inspect all background tasks using the Task Scheduler application. The tasks added by Git have names of the form *Git Maintenance (<frequency>)*. The Task Scheduler GUI has ways to inspect these tasks, but you can also export the tasks to XML files and view the details there.

Note that since Git is a console application, these background tasks create a console window visible to the current user. This can be changed manually by selecting the "Run whether user is logged in or not" option in Task Scheduler. This change requires a password input, which is why *git maintenance start* does not select it by default.

If you want to customize the background tasks, please rename the tasks so future calls to *git maintenance (start|stop)* do not overwrite your custom tasks.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

maintenance.auto

This boolean config option controls whether some commands run *git maintenance run --auto* after doing their normal work. Defaults to true.

maintenance.strategy

This string config option provides a way to specify one of a few recommended schedules for background maintenance. This only affects which tasks are run during *git maintenance run --schedule=X* commands, provided no *--task=<task>* arguments are provided. Further, if a *maintenance.<task>.schedule* config value is set, then that value is used instead of the one provided by *maintenance.strategy*. The possible strategy strings are:

- *none*: This default setting implies no tasks are run at any schedule.
- *incremental*: This setting optimizes for performing small maintenance activities that do not delete any data. This does not schedule the *gc* task, but runs the *prefetch* and *commit-graph* tasks hourly, the *loose-objects* and *incremental-repack* tasks daily, and the *pack-refs* task weekly.

maintenance.<task>.enabled

This boolean config option controls whether the maintenance task with name *<task>* is run when no *--task* option is specified to *git maintenance run*. These config values are ignored if a *--task* option exists. By default, only *maintenance.gc.enabled* is true.

maintenance.<task>.schedule

This config option controls whether or not the given *<task>* runs during a *git maintenance run --schedule=<frequency>* command. The value must be one of "hourly", "daily", or "weekly".

maintenance.commit-graph.auto

This integer config option controls how often the *commit-graph* task should be run as part of *git maintenance run --auto*. If zero, then the *commit-graph* task will not run with the *--auto* option. A negative value will force the task to run every time. Otherwise, a positive value implies the command should run when the number of reachable commits that are not in the commit-graph file is at least the value of *maintenance.commit-graph.auto*. The default value is 100.

maintenance.loose-objects.auto

This integer config option controls how often the *loose-objects* task should be run as part of *git maintenance run --auto*. If zero, then the *loose-objects* task will not run with the *--auto* option. A negative value will force

the task to run every time. Otherwise, a positive value implies the command should run when the number of loose objects is at least the value of *maintenance.loose-objects.auto*. The default value is 100.

maintenance.incremental-repack.auto

This integer config option controls how often the *incremental-repack* task should be run as part of *git maintenance run --auto*. If zero, then the *incremental-repack* task will not run with the *--auto* option. A negative value will force the task to run every time. Otherwise, a positive value implies the command should run when the number of pack-files not in the multi-pack-index is at least the value of *maintenance.incremental-repack.auto*. The default value is 10.

GIT

Part of the Section G.3.1, "git(1)" suite

G.3.81. git-merge-base(1)

2024-07-02
Revision History
2024-07-02

NAME

git-merge-base - Find as good common ancestors as possible for a merge

SYNOPSIS

git merge-base [-a | --all] <commit> <commit>…
git merge-base [-a | --all] --octopus <commit>…
git merge-base --is-ancestor <commit> <commit>
git merge-base --independent <commit>…
git merge-base --fork-point <ref> [<commit>]

DESCRIPTION

git merge-base finds the best common ancestor(s) between two commits to use in a three-way merge. One common ancestor is *better* than another common ancestor if the latter is an ancestor of the former. A common ancestor that does not have any better common ancestor is a *best common ancestor*, i.e. a *merge base*. Note that there can be more than one merge base for a pair of commits.

OPERATION MODES

In the most common special case, specifying only two commits on the command line means computing the merge base between the given two commits.

More generally, among the two commits to compute the merge base from, one is specified by the first commit argument on the command line; the other commit is a (possibly hypothetical) commit that is a merge across all the remaining commits on the command line.

As a consequence, the *merge base* is not necessarily contained in each of the commit arguments if more than two commits are specified. This is different from Section G.3.131, "git-show-branch(1)" when used with the *--merge-base* option.

--octopus

Compute the best common ancestors of all supplied commits, in preparation for an n-way merge. This mimics the behavior of *git show-branch --merge-base*.

--independent

Instead of printing merge bases, print a minimal subset of the supplied commits with the same ancestors. In other words, among the commits given, list those which cannot be reached from any other. This mimics the behavior of *git show-branch --independent*.

--is-ancestor

Check if the first <commit> is an ancestor of the second <commit>, and exit with status 0 if true, or with status 1 if not. Errors are signaled by a non-zero status that is not 1.

--fork-point

Find the point at which a branch (or any history that leads to <commit>) forked from another branch (or any reference) <ref>. This does not just look for the common ancestor of the two commits, but also takes into account the reflog of <ref> to see if the history leading to <commit> forked from an earlier incarnation of the branch <ref> (see discussion of this mode below).

OPTIONS

-a , --all

Output all merge bases for the commits, instead of just one.

DISCUSSION

Given two commits *A* and *B*, *git merge-base A B* will output a commit which is reachable from both *A* and *B* through the parent relationship.

For example, with this topology:

```
          o---o---o---B
         /
---o---1---o---o---o---A
```

the merge base between *A* and *B* is *1*.

Given three commits *A*, *B*, and *C*, *git merge-base A B C* will compute the merge base between *A* and a hypothetical commit *M*, which is a merge between *B* and *C*. For example, with this topology:

```
        o---o---o---o---C
       /
      /   o---o---o---B
     /   /
---2---1---o---o---o---A
```

the result of *git merge-base A B C* is *1*. This is because the equivalent topology with a merge commit *M* between *B* and *C* is:

```
        o---o---o---o---o
       /                 \
      /   o---o---o---o---M
     /   /
---2---1---o---o---o---A
```

and the result of *git merge-base A M* is *1*. Commit *2* is also a common ancestor between *A* and *M*, but *1* is a better common ancestor, because *2* is an ancestor of *1*. Hence, *2* is not a merge base.

The result of *git merge-base --octopus A B C* is *2*, because *2* is the best common ancestor of all commits.

When the history involves criss-cross merges, there can be more than one *best* common ancestor for two commits. For example, with this topology:

```
---1---o---A
    \ /
     X
    / \
---2---o---o---B
```

both *1* and *2* are merge bases of A and B. Neither one is better than the other (both are *best* merge bases). When the *--all* option is not given, it is unspecified which best one is output.

A common idiom to check "fast-forward-ness" between two commits A and B is (or at least used to be) to compute the merge base between A and B, and check if it is the same as A, in which case, A is an ancestor of B. You will see this idiom used often in older scripts.

```
A=$(git rev-parse --verify A)
if test "$A" = "$(git merge-base A B)"
then
        ... A is an ancestor of B ...
fi
```

In modern git, you can say this in a more direct way:

```
if git merge-base --is-ancestor A B
then
        ... A is an ancestor of B ...
fi
```

instead.

Discussion on fork-point mode

After working on the *topic* branch created with *git switch -c topic origin/master*, the history of remote-tracking branch *origin/master* may have been rewound and rebuilt, leading to a history of this shape:

```
                 o---B2
                /
---o---o---B1--o---o---o---B (origin/master)
        \
         B0
          \
           D0---D1---D (topic)
```

where *origin/master* used to point at commits B0, B1, B2 and now it points at B, and your *topic* branch was started on top of it back when *origin/master* was at B0, and you built three commits, D0, D1, and D, on top of it. Imagine that you now want to rebase the work you did on the topic on top of the updated origin/master.

In such a case, *git merge-base origin/master topic* would return the parent of B0 in the above picture, but B0^..D is **not** the range of commits you would want to replay on top of B (it includes B0, which is not what you wrote; it is a commit the other side discarded when it moved its tip from B0 to B1).

git merge-base --fork-point origin/master topic is designed to help in such a case. It takes not only B but also B0, B1, and B2 (i.e. old tips of the remote-tracking branches your repository's reflog knows about) into account to see on which commit your topic branch was built and finds B0, allowing you to replay only the commits on your topic, excluding the commits the other side later discarded.

Hence

```
$ fork_point=$(git merge-base --fork-point origin/master topic)
```

will find B0, and

```
$ git rebase --onto origin/master $fork_point topic
```

will replay D0, D1, and D on top of B to create a new history of this shape:

```
                 o---B2
                /
---o---o---B1--o---o---o---B (origin/master)
```

```
          \                    \
           B0                   D0'--D1'--D' (topic - updated)
            \
             D0---D1---D (topic - old)
```

A caveat is that older reflog entries in your repository may be expired by *git gc*. If B0 no longer appears in the reflog of the remote-tracking branch *origin/master*, the *--fork-point* mode obviously cannot find it and fails, avoiding to give a random and useless result (such as the parent of B0, like the same command without the *--fork-point* option gives).

Also, the remote-tracking branch you use the *--fork-point* mode with must be the one your topic forked from its tip. If you forked from an older commit than the tip, this mode would not find the fork point (imagine in the above sample history B0 did not exist, origin/master started at B1, moved to B2 and then B, and you forked your topic at origin/master^ when origin/master was B1; the shape of the history would be the same as above, without B0, and the parent of B1 is what *git merge-base origin/master topic* correctly finds, but the *--fork-point* mode will not, because it is not one of the commits that used to be at the tip of origin/master).

See also

Section G.3.120, “git-rev-list(1)”, Section G.3.131, “git-show-branch(1)”, Section G.3.86, “git-merge(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.82. git-merge-file(1)

2024-07-02
Revision History
2024-07-02

NAME

git-merge-file - Run a three-way file merge

SYNOPSIS

git merge-file [-L <current-name> [-L <base-name> [-L <other-name>]]]
[--ours|--theirs|--union] [-p|--stdout] [-q|--quiet] [--marker-size=<n>]
[--[no-]diff3] [--object-id] <current> <base> <other>

DESCRIPTION

Given three files *<current>*, *<base>* and *<other>*, *git merge-file* incorporates all changes that lead from *<base>* to *<other>* into *<current>*. The result ordinarily goes into *<current>*. *git merge-file* is useful for combining separate changes to an original. Suppose *<base>* is the original, and both *<current>* and *<other>* are modifications of *<base>*, then *git merge-file* combines both changes.

A conflict occurs if both *<current>* and *<other>* have changes in a common segment of lines. If a conflict is found, *git merge-file* normally outputs a warning and brackets the conflict with lines containing <<<<<<< and >>>>>>> markers. A typical conflict will look like this:

```
<<<<<<< A
lines in file A
=======
lines in file B
>>>>>>> B
```

If there are conflicts, the user should edit the result and delete one of the alternatives. When *--ours*, *--theirs*, or *--union* option is in effect, however, these conflicts are resolved favouring lines from *<current>*, lines from *<other>*, or lines from both respectively. The length of the conflict markers can be given with the *--marker-size* option.

If *--object-id* is specified, exactly the same behavior occurs, except that instead of specifying what to merge as files, it is specified as a list of object IDs referring to blobs.

The exit value of this program is negative on error, and the number of conflicts otherwise (truncated to 127 if there are more than that many conflicts). If the merge was clean, the exit value is 0.

git merge-file is designed to be a minimal clone of RCS *merge*; that is, it implements all of RCS *merge*'s functionality which is needed by Section G.3.1, “git(1)”.

OPTIONS

--object-id

Specify the contents to merge as blobs in the current repository instead of files. In this case, the operation must take place within a valid repository.

If the *-p* option is specified, the merged file (including conflicts, if any) goes to standard output as normal; otherwise, the merged file is written to the object store and the object ID of its blob is written to standard output.

-L <label>

This option may be given up to three times, and specifies labels to be used in place of the corresponding file names in conflict reports. That is, *git merge-file -L x -L y -L z a b c* generates output that looks like it came from files x, y and z instead of from files a, b and c.

-p

Send results to standard output instead of overwriting *<current>*.

-q

Quiet; do not warn about conflicts.

--diff3

Show conflicts in "diff3" style.

--zdiff3

Show conflicts in "zdiff3" style.

--ours , --theirs , --union

Instead of leaving conflicts in the file, resolve conflicts favouring our (or their or both) side of the lines.

--diff-algorithm={patience|minimal|histogram|myers}

Use a different diff algorithm while merging. The current default is "myers", but selecting more recent algorithm such as "histogram" can help avoid mismerges that occur due to unimportant matching lines (such as braces from distinct functions). See also Section G.3.44, “git-diff(1)” *--diff-algorithm*.

EXAMPLES

git merge-file README.my README README.upstream

combines the changes of README.my and README.upstream since README, tries to merge them and writes the result into README.my.

git merge-file -L a -L b -L c tmp/a123 tmp/b234 tmp/c345

merges tmp/a123 and tmp/c345 with the base tmp/b234, but uses labels *a* and *c* instead of *tmp/a123* and *tmp/c345*.

git merge-file -p --object-id abc1234 def567 890abcd

combines the changes of the blob abc1234 and 890abcd since def567, tries to merge them and writes the result to standard output

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.83. git-merge-index(1)

2024-07-02
Revision History
2024-07-02

NAME

git-merge-index - Run a merge for files needing merging

SYNOPSIS

git merge-index [-o] [-q] <merge-program> (-a | ([--] <file>…))

DESCRIPTION

This looks up the <file>(s) in the index and, if there are any merge entries, passes the SHA-1 hash for those files as arguments 1, 2, 3 (empty argument if no file), and <file> as argument 4. File modes for the three files are passed as arguments 5, 6 and 7.

OPTIONS

--

Do not interpret any more arguments as options.

-a

Run merge against all files in the index that need merging.

-o

Instead of stopping at the first failed merge, do all of them in one shot - continue with merging even when previous merges returned errors, and only return the error code after all the merges.

-q

Do not complain about a failed merge program (a merge program failure usually indicates conflicts during the merge). This is for porcelains which might want to emit custom messages.

If *git merge-index* is called with multiple <file>s (or -a) then it processes them in turn only stopping if merge returns a non-zero exit code.

Typically this is run with a script calling Git's imitation of the *merge* command from the RCS package.

A sample script called *git merge-one-file* is included in the distribution.

ALERT ALERT ALERT! The Git "merge object order" is different from the RCS *merge* program merge object order. In the above ordering, the original is first. But the argument order to the 3-way merge program *merge* is to have the original in the middle. Don't ask me why.

Examples:

```
torvalds@ppc970:~/merge-test> git merge-index cat MM
This is MM from the original tree.              # original
```

```
This is modified MM in the branch A.            # merge1
This is modified MM in the branch B.            # merge2
This is modified MM in the branch B.            # current contents
```

or

```
torvalds@ppc970:~/merge-test> git merge-index cat AA MM
cat: : No such file or directory
This is added AA in the branch A.
This is added AA in the branch B.
This is added AA in the branch B.
fatal: merge program failed
```

where the latter example shows how *git merge-index* will stop trying to merge once anything has returned an error (i.e., *cat* returned an error for the AA file, because it didn't exist in the original, and thus *git merge-index* didn't even try to merge the MM thing).

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.84. git-merge-one-file(1)

2024-07-02
Revision History
2024-07-02

NAME

git-merge-one-file - The standard helper program to use with git-merge-index

SYNOPSIS

git merge-one-file

DESCRIPTION

This is the standard helper program to use with *git merge-index* to resolve a merge after the trivial merge done with *git read-tree -m*.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.85. git-merge-tree(1)

2024-07-02
Revision History
2024-07-02

NAME

git-merge-tree - Perform merge without touching index or working tree

SYNOPSIS

git merge-tree [--write-tree] [<options>] <branch1> <branch2>
git merge-tree [--trivial-merge] <base-tree> <branch1> <branch2> (deprecated)

DESCRIPTION

This command has a modern *--write-tree* mode and a deprecated *--trivial-merge* mode. With the exception of the DEPRECATED DESCRIPTION section at the end, the rest of this documentation describes the modern *--write-tree* mode.

Performs a merge, but does not make any new commits and does not read from or write to either the working tree or index.

The performed merge will use the same features as the "real" Section G.3.86, “git-merge(1)”, including:

- three way content merges of individual files
- rename detection
- proper directory/file conflict handling
- recursive ancestor consolidation (i.e. when there is more than one merge base, creating a virtual merge base by merging the merge bases)
- etc.

After the merge completes, a new toplevel tree object is created. See *OUTPUT* below for details.

OPTIONS

-z

Do not quote filenames in the <Conflicted file info> section, and end each filename with a NUL character rather than newline. Also begin the messages section with a NUL character instead of a newline. See the section called “OUTPUT” below for more information.

--name-only

In the Conflicted file info section, instead of writing a list of (mode, oid, stage, path) tuples to output for conflicted files, just provide a list of filenames with conflicts (and do not list filenames multiple times if they have multiple conflicting stages).

--[no-]messages

Write any informational messages such as "Auto-merging <path>" or CONFLICT notices to the end of stdout. If unspecified, the default is to include these messages if there are merge conflicts, and to omit them otherwise.

--allow-unrelated-histories

merge-tree will by default error out if the two branches specified share no common history. This flag can be given to override that check and make the merge proceed anyway.

--merge-base=<tree-ish>

Instead of finding the merge-bases for <branch1> and <branch2>, specify a merge-base for the merge, and specifying multiple bases is currently not supported. This option is incompatible with *--stdin*.

As the merge-base is provided directly, <branch1> and <branch2> do not need to specify commits; trees are enough.

-X<option> , --strategy-option=<option>

Pass the merge strategy-specific option through to the merge strategy. See Section G.3.86, “git-merge(1)” for details.

OUTPUT

For a successful merge, the output from git-merge-tree is simply one line:

```
<OID of toplevel tree>
```

Whereas for a conflicted merge, the output is by default of the form:

```
<OID of toplevel tree>
<Conflicted file info>
<Informational messages>
```

These are discussed individually below.

However, there is an exception. If *--stdin* is passed, then there is an extra section at the beginning, a NUL character at the end, and then all the sections repeat for each line of input. Thus, if the first merge is conflicted and the second is clean, the output would be of the form:

```
<Merge status>
<OID of toplevel tree>
<Conflicted file info>
<Informational messages>
NUL
<Merge status>
<OID of toplevel tree>
NUL
```

1. Merge status

This is an integer status followed by a NUL character. The integer status is:

```
0: merge had conflicts
1: merge was clean
<0: something prevented the merge from running (e.g. access to repository
    objects denied by filesystem)
```

2. OID of toplevel tree

This is a tree object that represents what would be checked out in the working tree at the end of *git merge*. If there were conflicts, then files within this tree may have embedded conflict markers. This section is always followed by a newline (or NUL if *-z* is passed).

3. Conflicted file info

This is a sequence of lines with the format

```
<mode> <object> <stage> <filename>
```

The filename will be quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). However, if the *--name-only* option is passed, the mode, object, and stage will be omitted. If *-z* is passed, the "lines" are terminated by a NUL character instead of a newline character.

4. Informational messages

This section provides informational messages, typically about conflicts. The format of the section varies significantly depending on whether *-z* is passed.

If *-z* is passed:

The output format is zero or more conflict informational records, each of the form:

```
<list-of-paths><conflict-type>NUL<conflict-message>NUL
```

where <list-of-paths> is of the form

```
<number-of-paths>NUL<path1>NUL<path2>NUL...<pathN>NUL
```

and includes paths (or branch names) affected by the conflict or informational message in <conflict-message>. Also, <conflict-type> is a stable string explaining the type of conflict, such as

- "Auto-merging"
- "CONFLICT (rename/delete)"
- "CONFLICT (submodule lacks merge base)"
- "CONFLICT (binary)"

and <conflict-message> is a more detailed message about the conflict which often (but not always) embeds the <stable-short-type-description> within it. These strings may change in future Git versions. Some examples:

- "Auto-merging <file>"
- "CONFLICT (rename/delete): <oldfile> renamed…but deleted in…"
- "Failed to merge submodule <submodule> (no merge base)"
- "Warning: cannot merge binary files: <filename>"

If *-z* is NOT passed:

This section starts with a blank line to separate it from the previous sections, and then only contains the <conflict-message> information from the previous section (separated by newlines). These are non-stable strings that should not be parsed by scripts, and are just meant for human consumption. Also, note that while <conflict-message> strings usually do not contain embedded newlines, they sometimes do. (However, the free-form messages will never have an embedded NUL character). So, the entire block of information is meant for human readers as an agglomeration of all conflict messages.

EXIT STATUS

For a successful, non-conflicted merge, the exit status is 0. When the merge has conflicts, the exit status is 1. If the merge is not able to complete (or start) due to some kind of error, the exit status is something other than 0 or 1 (and the output is unspecified). When --stdin is passed, the return status is 0 for both successful and conflicted merges, and something other than 0 or 1 if it cannot complete all the requested merges.

USAGE NOTES

This command is intended as low-level plumbing, similar to Section G.3.62, “git-hash-object(1)”, Section G.3.90, “git-mktree(1)”, Section G.3.27, “git-commit-tree(1)”, Section G.3.160, “git-write-tree(1)”, Section G.3.148, “git-update-ref(1)”, and Section G.3.89, “git-mktag(1)”. Thus, it can be used as a part of a series of steps such as:

```
NEWTREE=$(git merge-tree --write-tree $BRANCH1 $BRANCH2)
test $? -eq 0 || die "There were conflicts..."
NEWCOMMIT=$(git commit-tree $NEWTREE -p $BRANCH1 -p $BRANCH2)
git update-ref $BRANCH1 $NEWCOMMIT
```

Note that when the exit status is non-zero, *NEWTREE* in this sequence will contain a lot more output than just a tree.

For conflicts, the output includes the same information that you'd get with Section G.3.86, “git-merge(1)”:

- what would be written to the working tree (the OID of toplevel tree)
- the higher order stages that would be written to the index (the Conflicted file info)
- any messages that would have been printed to stdout (the Informational messages)

INPUT FORMAT

git merge-tree --stdin input format is fully text based. Each line has this format:

```
[<base-commit> -- ]<branch1> <branch2>
```

If one line is separated by --, the string before the separator is used for specifying a merge-base for the merge and the string after the separator describes the branches to be merged.

MISTAKES TO AVOID

Do NOT look through the resulting toplevel tree to try to find which files conflict; parse the Conflicted file info section instead. Not only would parsing an entire tree be horrendously slow in large repositories, there are numerous types of conflicts not representable by conflict markers (modify/delete, mode conflict, binary file changed on both sides, file/directory conflicts, various rename conflict permutations, etc.)

Do NOT interpret an empty Conflicted file info list as a clean merge; check the exit status. A merge can have conflicts without having individual files conflict (there are a few types of directory rename conflicts that fall into this category, and others might also be added in the future).

Do NOT attempt to guess or make the user guess the conflict types from the Conflicted file info list. The information there is insufficient to do so. For example: Rename/rename(1to2) conflicts (both sides renamed the same file differently) will result in three different files having higher order stages (but each only has one higher order stage), with no way (short of the Informational messages section) to determine which three files are related. File/directory conflicts also result in a file with exactly one higher order stage. Possibly-involved-in-directory-rename conflicts (when "merge.directoryRenames" is unset or set to "conflicts") also result in a file with exactly one higher order stage. In all cases, the Informational messages section has the necessary info, though it is not designed to be machine parseable.

Do NOT assume that each path from Conflicted file info, and the logical conflicts in the Informational messages have a one-to-one mapping, nor that there is a one-to-many mapping, nor a many-to-one mapping. Many-to-many mappings exist, meaning that each path can have many logical conflict types in a single merge, and each logical conflict type can affect many paths.

Do NOT assume all filenames listed in the Informational messages section had conflicts. Messages can be included for files that have no conflicts, such as "Auto-merging <file>".

AVOID taking the OIDS from the Conflicted file info and re-merging them to present the conflicts to the user. This will lose information. Instead, look up the version of the file found within the OID of toplevel tree and show that instead. In particular, the latter will have conflict markers annotated with the original branch/commit being merged and, if renames were involved, the original filename. While you could include the original branch/commit in the conflict marker annotations when re-merging, the original filename is not available from the Conflicted file info and thus you would be losing information that might help the user resolve the conflict.

DEPRECATED DESCRIPTION

Per the DESCRIPTION and unlike the rest of this documentation, this section describes the deprecated *--trivial-merge* mode.

Other than the optional *--trivial-merge*, this mode accepts no options.

This mode reads three tree-ish, and outputs trivial merge results and conflicting stages to the standard output in a semi-diff format. Since this was designed for higher level scripts to consume and merge the results back into the index, it omits entries that match <branch1>. The result of this second form is similar to what three-way *git read-tree -m* does, but instead of storing the results in the index, the command outputs the entries to the standard output.

This form not only has limited applicability (a trivial merge cannot handle content merges of individual files, rename detection, proper directory/file conflict handling, etc.), the output format is also difficult to work with, and it will generally be less performant than the first form even on successful merges (especially if working in large repositories).

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.86. git-merge(1)

2024-07-02

Revision History

2024-07-02

NAME

git-merge - Join two or more development histories together

SYNOPSIS

git merge [-n] [--stat] [--no-commit] [--squash] [--[no-]edit]
[--no-verify] [-s <strategy>] [-X <strategy-option>] [-S[<keyid>]]
[--[no-]allow-unrelated-histories]
[--[no-]rerere-autoupdate] [-m <msg>] [-F <file>]
[--into-name <branch>] [<commit>…]
git merge (--continue | --abort | --quit)

DESCRIPTION

Incorporates changes from the named commits (since the time their histories diverged from the current branch) into the current branch. This command is used by *git pull* to incorporate changes from another repository and can be used by hand to merge changes from one branch into another.

Assume the following history exists and the current branch is *master*:

```
          A---B---C topic
         /
    D---E---F---G master
```

Then *git merge topic* will replay the changes made on the *topic* branch since it diverged from *master* (i.e., *E*) until its current commit (*C*) on top of *master*, and record the result in a new commit along with the names of the two parent commits and a log message from the user describing the changes. Before the operation, *ORIG_HEAD* is set to the tip of the current branch (*C*).

```
          A---B---C topic
         /         \
    D---E---F---G---H master
```

A merge stops if there's a conflict that cannot be resolved automatically or if *--no-commit* was provided when initiating the merge. At that point you can run *git merge --abort* or *git merge --continue*.

git merge --abort will abort the merge process and try to reconstruct the pre-merge state. However, if there were uncommitted changes when the merge started (and especially if those changes were further modified after the merge was started), *git merge --abort* will in some cases be unable to reconstruct the original (pre-merge) changes. Therefore:

Warning: Running *git merge* with non-trivial uncommitted changes is discouraged: while possible, it may leave you in a state that is hard to back out of in the case of a conflict.

OPTIONS

--commit , --no-commit

Perform the merge and commit the result. This option can be used to override --no-commit.

With --no-commit perform the merge and stop just before creating a merge commit, to give the user a chance to inspect and further tweak the merge result before committing.

Note that fast-forward updates do not create a merge commit and therefore there is no way to stop those merges with --no-commit. Thus, if you want to ensure your branch is not changed or updated by the merge command, use --no-ff with --no-commit.

--edit , -e , --no-edit

Invoke an editor before committing successful mechanical merge to further edit the auto-generated merge message, so that the user can explain and justify the merge. The *--no-edit* option can be used to accept the auto-generated message (this is generally discouraged). The *--edit* (or *-e*) option is still useful if you are giving a draft message with the *-m* option from the command line and want to edit it in the editor.

Older scripts may depend on the historical behaviour of not allowing the user to edit the merge log message. They will see an editor opened when they run *git merge*. To make it easier to adjust such scripts to the updated behaviour, the environment variable *GIT_MERGE_AUTOEDIT* can be set to *no* at the beginning of them.

--cleanup=<mode>

This option determines how the merge message will be cleaned up before committing. See Section G.3.28, "git-commit(1)" for more details. In addition, if the *<mode>* is given a value of *scissors*, scissors will be appended to *MERGE_MSG* before being passed on to the commit machinery in the case of a merge conflict.

--ff , --no-ff , --ff-only

Specifies how a merge is handled when the merged-in history is already a descendant of the current history. *--ff* is the default unless merging an annotated (and possibly signed) tag that is not stored in its natural place in the *refs/tags/* hierarchy, in which case *--no-ff* is assumed.

With *--ff*, when possible resolve the merge as a fast-forward (only update the branch pointer to match the merged branch; do not create a merge commit). When not possible (when the merged-in history is not a descendant of the current history), create a merge commit.

With *--no-ff*, create a merge commit in all cases, even when the merge could instead be resolved as a fast-forward.

With *--ff-only*, resolve the merge as a fast-forward when possible. When not possible, refuse to merge and exit with a non-zero status.

-S[<keyid>] , --gpg-sign[=<keyid>] , --no-gpg-sign

GPG-sign the resulting merge commit. The *keyid* argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space. *--no-gpg-sign* is useful to countermand both *commit.gpgSign* configuration variable, and earlier *--gpg-sign*.

--log[=<n>] , --no-log

In addition to branch names, populate the log message with one-line descriptions from at most <n> actual commits that are being merged. See also Section G.3.51, "git-fmt-merge-msg(1)".

With --no-log do not list one-line descriptions from the actual commits being merged.

--signoff , --no-signoff

Add a *Signed-off-by* trailer by the committer at the end of the commit log message. The meaning of a signoff depends on the project to which you're committing. For example, it may certify that the committer has the rights to submit the work under the project's license or agrees to some contributor representation, such as a Developer Certificate of Origin. (See https://developercertificate.org for the one used by the Linux kernel and Git projects.) Consult the documentation or leadership of the project to which you're contributing to understand how the signoffs are used in that project.

The --no-signoff option can be used to countermand an earlier --signoff option on the command line.

--stat , -n , --no-stat

Show a diffstat at the end of the merge. The diffstat is also controlled by the configuration option merge.stat.

With -n or --no-stat do not show a diffstat at the end of the merge.

--squash , --no-squash

Produce the working tree and index state as if a real merge happened (except for the merge information), but do not actually make a commit, move the *HEAD*, or record *$GIT_DIR/MERGE_HEAD* (to cause the next *git commit* command to create a merge commit). This allows you to create a single commit on top of the current branch whose effect is the same as merging another branch (or more in case of an octopus).

With --no-squash perform the merge and commit the result. This option can be used to override --squash.

With --squash, --commit is not allowed, and will fail.

--[no-]verify

By default, the pre-merge and commit-msg hooks are run. When *--no-verify* is given, these are bypassed. See also Section G.4.7, “githooks(5)”.

-s <strategy> , --strategy=<strategy>

Use the given merge strategy; can be supplied more than once to specify them in the order they should be tried. If there is no *-s* option, a built-in list of strategies is used instead (*ort* when merging a single head, *octopus* otherwise).

-X <option> , --strategy-option=<option>

Pass merge strategy specific option through to the merge strategy.

--verify-signatures , --no-verify-signatures

Verify that the tip commit of the side branch being merged is signed with a valid key, i.e. a key that has a valid uid: in the default trust model, this means the signing key has been signed by a trusted key. If the tip commit of the side branch is not signed with a valid key, the merge is aborted.

--summary , --no-summary

Synonyms to --stat and --no-stat; these are deprecated and will be removed in the future.

-q , --quiet

Operate quietly. Implies --no-progress.

-v , --verbose

Be verbose.

--progress , --no-progress

Turn progress on/off explicitly. If neither is specified, progress is shown if standard error is connected to a terminal. Note that not all merge strategies may support progress reporting.

--autostash , --no-autostash

Automatically create a temporary stash entry before the operation begins, record it in the ref *MERGE_AUTOSTASH* and apply it after the operation ends. This means that you can run the operation on a dirty worktree. However, use with care: the final stash application after a successful merge might result in non-trivial conflicts.

--allow-unrelated-histories

By default, *git merge* command refuses to merge histories that do not share a common ancestor. This option can be used to override this safety when merging histories of two projects that started their lives independently. As that is a very rare occasion, no configuration variable to enable this by default exists and will not be added.

-m <msg>

Set the commit message to be used for the merge commit (in case one is created).

If --*log* is specified, a shortlog of the commits being merged will be appended to the specified message.

The *git fmt-merge-msg* command can be used to give a good default for automated *git merge* invocations. The automated message can include the branch description.

--into-name <branch>

Prepare the default merge message as if merging to the branch <*branch*>, instead of the name of the real branch to which the merge is made.

-F <file> , --file=<file>

Read the commit message to be used for the merge commit (in case one is created).

If --*log* is specified, a shortlog of the commits being merged will be appended to the specified message.

--rerere-autoupdate , --no-rerere-autoupdate

After the rerere mechanism reuses a recorded resolution on the current conflict to update the files in the working tree, allow it to also update the index with the result of resolution. --*no-rerere-autoupdate* is a good way to double-check what *rerere* did and catch potential mismerges, before committing the result to the index with a separate *git add*.

--overwrite-ignore , --no-overwrite-ignore

Silently overwrite ignored files from the merge result. This is the default behavior. Use --*no-overwrite-ignore* to abort.

--abort

Abort the current conflict resolution process, and try to reconstruct the pre-merge state. If an autostash entry is present, apply it to the worktree.

If there were uncommitted worktree changes present when the merge started, *git merge* --*abort* will in some cases be unable to reconstruct these changes. It is therefore recommended to always commit or stash your changes before running *git merge*.

git merge --*abort* is equivalent to *git reset* --*merge* when *MERGE_HEAD* is present unless *MERGE_AUTOSTASH* is also present in which case *git merge* --*abort* applies the stash entry to the worktree whereas *git reset* --*merge* will save the stashed changes in the stash list.

--quit

Forget about the current merge in progress. Leave the index and the working tree as-is. If *MERGE_AUTOSTASH* is present, the stash entry will be saved to the stash list.

--continue

After a *git merge* stops due to conflicts you can conclude the merge by running *git merge* --*continue* (see "HOW TO RESOLVE CONFLICTS" section below).

<commit>…

Commits, usually other branch heads, to merge into our branch. Specifying more than one commit will create a merge with more than two parents (affectionately called an Octopus merge).

If no commit is given from the command line, merge the remote-tracking branches that the current branch is configured to use as its upstream. See also the configuration section of this manual page.

When *FETCH_HEAD* (and no other commit) is specified, the branches recorded in the *.git/FETCH_HEAD* file by the previous invocation of *git fetch* for merging are merged to the current branch.

PRE-MERGE CHECKS

Before applying outside changes, you should get your own work in good shape and committed locally, so it will not be clobbered if there are conflicts. See also Section G.3.137, “git-stash(1)”. *git pull* and *git merge* will stop without doing anything when local uncommitted changes overlap with files that *git pull*/*git merge* may need to update.

To avoid recording unrelated changes in the merge commit, *git pull* and *git merge* will also abort if there are any changes registered in the index relative to the *HEAD* commit. (Special narrow exceptions to this rule may exist depending on which merge strategy is in use, but generally, the index must match HEAD.)

If all named commits are already ancestors of *HEAD*, *git merge* will exit early with the message "Already up to date."

FAST-FORWARD MERGE

Often the current branch head is an ancestor of the named commit. This is the most common case especially when invoked from *git pull*: you are tracking an upstream repository, you have committed no local changes, and now you want to update to a newer upstream revision. In this case, a new commit is not needed to store the combined history; instead, the *HEAD* (along with the index) is updated to point at the named commit, without creating an extra merge commit.

This behavior can be suppressed with the *--no-ff* option.

TRUE MERGE

Except in a fast-forward merge (see above), the branches to be merged must be tied together by a merge commit that has both of them as its parents.

A merged version reconciling the changes from all branches to be merged is committed, and your *HEAD*, index, and working tree are updated to it. It is possible to have modifications in the working tree as long as they do not overlap; the update will preserve them.

When it is not obvious how to reconcile the changes, the following happens:

1. The *HEAD* pointer stays the same.
2. The *MERGE_HEAD* ref is set to point to the other branch head.
3. Paths that merged cleanly are updated both in the index file and in your working tree.
4. For conflicting paths, the index file records up to three versions: stage 1 stores the version from the common ancestor, stage 2 from *HEAD*, and stage 3 from *MERGE_HEAD* (you can inspect the stages with *git ls-files -u*). The working tree files contain the result of the merge operation; i.e. 3-way merge results with familiar conflict markers <<< === >>>.
5. A ref named *AUTO_MERGE* is written, pointing to a tree corresponding to the current content of the working tree (including conflict markers for textual conflicts). Note that this ref is only written when the *ort* merge strategy is used (the default).
6. No other changes are made. In particular, the local modifications you had before you started merge will stay the same and the index entries for them stay as they were, i.e. matching *HEAD*.

If you tried a merge which resulted in complex conflicts and want to start over, you can recover with *git merge --abort*.

MERGING TAG

When merging an annotated (and possibly signed) tag, Git always creates a merge commit even if a fast-forward merge is possible, and the commit message template is prepared with the tag message. Additionally, if the tag is signed, the signature check is reported as a comment in the message template. See also Section G.3.144, “git-tag(1)”.

When you want to just integrate with the work leading to the commit that happens to be tagged, e.g. synchronizing with an upstream release point, you may not want to make an unnecessary merge commit.

In such a case, you can "unwrap" the tag yourself before feeding it to *git merge*, or pass *--ff-only* when you do not have any work on your own. e.g.

```
git fetch origin
git merge v1.2.3^0
git merge --ff-only v1.2.3
```

HOW CONFLICTS ARE PRESENTED

During a merge, the working tree files are updated to reflect the result of the merge. Among the changes made to the common ancestor's version, non-overlapping ones (that is, you changed an area of the file while the other side left that area intact, or vice versa) are incorporated in the final result verbatim. When both sides made changes to the same area, however, Git cannot randomly pick one side over the other, and asks you to resolve it by leaving what both sides did to that area.

By default, Git uses the same style as the one used by the "merge" program from the RCS suite to present such a conflicted hunk, like this:

```
Here are lines that are either unchanged from the common
ancestor, or cleanly resolved because only one side changed,
or cleanly resolved because both sides changed the same way.
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodified.
```

The area where a pair of conflicting changes happened is marked with markers `<<<<<<<`, `=======`, and `>>>>>>>`. The part before the `=======` is typically your side, and the part afterwards is typically their side.

The default format does not show what the original said in the conflicting area. You cannot tell how many lines are deleted and replaced with Barbie's remark on your side. The only thing you can tell is that your side wants to say it is hard and you'd prefer to go shopping, while the other side wants to claim it is easy.

An alternative style can be used by setting the *merge.conflictStyle* configuration variable to either "diff3" or "zdiff3". In "diff3" style, the above conflict may look like this:

```
Here are lines that are either unchanged from the common
ancestor, or cleanly resolved because only one side changed,
<<<<<<< yours:sample.txt
or cleanly resolved because both sides changed the same way.
Conflict resolution is hard;
let's go shopping.
||||||| base:sample.txt
or cleanly resolved because both sides changed identically.
Conflict resolution is hard.
=======
or cleanly resolved because both sides changed the same way.
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodified.
```

while in "zdiff3" style, it may look like this:

```
Here are lines that are either unchanged from the common
```

```
ancestor, or cleanly resolved because only one side changed,
or cleanly resolved because both sides changed the same way.
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
||||||| base:sample.txt
or cleanly resolved because both sides changed identically.
Conflict resolution is hard.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodified.
```

In addition to the `<<<<<<<`, `=======`, and `>>>>>>>` markers, it uses another ||||||| marker that is followed by the original text. You can tell that the original just stated a fact, and your side simply gave in to that statement and gave up, while the other side tried to have a more positive attitude. You can sometimes come up with a better resolution by viewing the original.

HOW TO RESOLVE CONFLICTS

After seeing a conflict, you can do two things:

- Decide not to merge. The only clean-ups you need are to reset the index file to the *HEAD* commit to reverse 2. and to clean up working tree changes made by 2. and 3.; *git merge --abort* can be used for this.
- Resolve the conflicts. Git will mark the conflicts in the working tree. Edit the files into shape and *git add* them to the index. Use *git commit* or *git merge --continue* to seal the deal. The latter command checks whether there is a (interrupted) merge in progress before calling *git commit*.

You can work through the conflict with a number of tools:

- Use a mergetool. *git mergetool* to launch a graphical mergetool which will work through the merge with you.
- Look at the diffs. *git diff* will show a three-way diff, highlighting changes from both the *HEAD* and *MERGE_HEAD* versions. *git diff AUTO_MERGE* will show what changes you've made so far to resolve textual conflicts.
- Look at the diffs from each branch. *git log --merge -p <path>* will show diffs first for the *HEAD* version and then the *MERGE_HEAD* version.
- Look at the originals. *git show :1:filename* shows the common ancestor, *git show :2:filename* shows the *HEAD* version, and *git show :3:filename* shows the *MERGE_HEAD* version.

EXAMPLES

- Merge branches *fixes* and *enhancements* on top of the current branch, making an octopus merge:

  ```
  $ git merge fixes enhancements
  ```

- Merge branch *obsolete* into the current branch, using *ours* merge strategy:

  ```
  $ git merge -s ours obsolete
  ```

- Merge branch *maint* into the current branch, but do not make a new commit automatically:

  ```
  $ git merge --no-commit maint
  ```

  This can be used when you want to include further changes to the merge, or want to write your own merge commit message.

  You should refrain from abusing this option to sneak substantial changes into a merge commit. Small fixups like bumping release/version name would be acceptable.

MERGE STRATEGIES

The merge mechanism (*git merge* and *git pull* commands) allows the backend *merge strategies* to be chosen with *-s* option. Some strategies can also take their own options, which can be passed by giving *-X<option>* arguments to *git merge* and/or *git pull*.

ort

This is the default merge strategy when pulling or merging one branch. This strategy can only resolve two heads using a 3-way merge algorithm. When there is more than one common ancestor that can be used for 3-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the 3-way merge. This has been reported to result in fewer merge conflicts without causing mismerges by tests done on actual merge commits taken from Linux 2.6 kernel development history. Additionally this strategy can detect and handle merges involving renames. It does not make use of detected copies. The name for this algorithm is an acronym ("Ostensibly Recursive's Twin") and came from the fact that it was written as a replacement for the previous default algorithm, *recursive*.

The *ort* strategy can take the following options:

ours

This option forces conflicting hunks to be auto-resolved cleanly by favoring *our* version. Changes from the other tree that do not conflict with our side are reflected in the merge result. For a binary file, the entire contents are taken from our side.

This should not be confused with the *ours* merge strategy, which does not even look at what the other tree contains at all. It discards everything the other tree did, declaring *our* history contains all that happened in it.

theirs

This is the opposite of *ours*; note that, unlike *ours*, there is no *theirs* merge strategy to confuse this merge option with.

ignore-space-change , ignore-all-space , ignore-space-at-eol , ignore-cr-at-eol

Treats lines with the indicated type of whitespace change as unchanged for the sake of a three-way merge. Whitespace changes mixed with other changes to a line are not ignored. See also Section G.3.44, “git-diff(1)” *-b*, *-w*, *--ignore-space-at-eol*, and *--ignore-cr-at-eol*.

- If *their* version only introduces whitespace changes to a line, *our* version is used;
- If *our* version introduces whitespace changes but *their* version includes a substantial change, *their* version is used;
- Otherwise, the merge proceeds in the usual way.

renormalize

This runs a virtual check-out and check-in of all three stages of a file when resolving a three-way merge. This option is meant to be used when merging branches with different clean filters or end-of-line normalization rules. See "Merging branches with differing checkin/checkout attributes" in Section G.4.2, “gitattributes(5)” for details.

no-renormalize

Disables the *renormalize* option. This overrides the *merge.renormalize* configuration variable.

find-renames[=<n>]

Turn on rename detection, optionally setting the similarity threshold. This is the default. This overrides the *merge.renames* configuration variable. See also Section G.3.44, “git-diff(1)” *--find-renames*.

rename-threshold=<n>

Deprecated synonym for *find-renames=<n>*.

subtree[=<path>]

This option is a more advanced form of *subtree* strategy, where the strategy makes a guess on how two trees must be shifted to match with each other when merging. Instead, the specified path is prefixed (or stripped from the beginning) to make the shape of two trees to match.

recursive

This can only resolve two heads using a 3-way merge algorithm. When there is more than one common ancestor that can be used for 3-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the 3-way merge. This has been reported to result in fewer merge conflicts without causing mismerges by tests done on actual merge commits taken from Linux 2.6 kernel development history. Additionally this can detect and handle merges involving renames. It does not make use of detected copies. This was the default strategy for resolving two heads from Git v0.99.9k until v2.33.0.

The *recursive* strategy takes the same options as *ort*. However, there are three additional options that *ort* ignores (not documented above) that are potentially useful with the *recursive* strategy:

patience

Deprecated synonym for *diff-algorithm=patience*.

diff-algorithm=[patience|minimal|histogram|myers]

Use a different diff algorithm while merging, which can help avoid mismerges that occur due to unimportant matching lines (such as braces from distinct functions). See also Section G.3.44, “git-diff(1)” *--diff-algorithm*. Note that *ort* specifically uses *diff-algorithm=histogram*, while *recursive* defaults to the *diff.algorithm* config setting.

no-renames

Turn off rename detection. This overrides the *merge.renames* configuration variable. See also Section G.3.44, “git-diff(1)” *--no-renames*.

resolve

This can only resolve two heads (i.e. the current branch and another branch you pulled from) using a 3-way merge algorithm. It tries to carefully detect criss-cross merge ambiguities. It does not handle renames.

octopus

This resolves cases with more than two heads, but refuses to do a complex merge that needs manual resolution. It is primarily meant to be used for bundling topic branch heads together. This is the default merge strategy when pulling or merging more than one branch.

ours

This resolves any number of heads, but the resulting tree of the merge is always that of the current branch head, effectively ignoring all changes from all other branches. It is meant to be used to supersede old development history of side branches. Note that this is different from the -Xours option to the *recursive* merge strategy.

subtree

This is a modified *ort* strategy. When merging trees A and B, if B corresponds to a subtree of A, B is first adjusted to match the tree structure of A, instead of reading the trees at the same level. This adjustment is also done to the common ancestor tree.

With the strategies that use 3-way merge (including the default, *ort*), if a change is made on both branches, but later reverted on one of the branches, that change will be present in the merged result; some people find this behavior

confusing. It occurs because only the heads and the merge base are considered when performing a merge, not the individual commits. The merge algorithm therefore considers the reverted change as no change at all, and substitutes the changed version instead.

CONFIGURATION

branch.<name>.mergeOptions

Sets default options for merging into branch <name>. The syntax and supported options are the same as those of *git merge*, but option values containing whitespace characters are currently not supported.

Everything above this line in this section isn't included from the Section G.3.29, “git-config(1)” documentation. The content that follows is the same as what's found there:

merge.conflictStyle

Specify the style in which conflicted hunks are written out to working tree files upon merge. The default is "merge", which shows a `<<<<<<<` conflict marker, changes made by one side, a `=======` marker, changes made by the other side, and then a `>>>>>>>` marker. An alternate style, "diff3", adds a `|||||||` marker and the original text before the `=======` marker. The "merge" style tends to produce smaller conflict regions than diff3, both because of the exclusion of the original text, and because when a subset of lines match on the two sides, they are just pulled out of the conflict region. Another alternate style, "zdiff3", is similar to diff3 but removes matching lines on the two sides from the conflict region when those matching lines appear near either the beginning or end of a conflict region.

merge.defaultToUpstream

If merge is called without any commit argument, merge the upstream branches configured for the current branch by using their last observed values stored in their remote-tracking branches. The values of the *branch.<current branch>.merge* that name the branches at the remote named by *branch.<current branch>.remote* are consulted, and then they are mapped via *remote.<remote>.fetch* to their corresponding remote-tracking branches, and the tips of these tracking branches are merged. Defaults to true.

merge.ff

By default, Git does not create an extra merge commit when merging a commit that is a descendant of the current commit. Instead, the tip of the current branch is fast-forwarded. When set to *false*, this variable tells Git to create an extra merge commit in such a case (equivalent to giving the *--no-ff* option from the command line). When set to *only*, only such fast-forward merges are allowed (equivalent to giving the *--ff-only* option from the command line).

merge.verifySignatures

If true, this is equivalent to the --verify-signatures command line option. See Section G.3.86, “git-merge(1)” for details.

merge.branchdesc

In addition to branch names, populate the log message with the branch description text associated with them. Defaults to false.

merge.log

In addition to branch names, populate the log message with at most the specified number of one-line descriptions from the actual commits that are being merged. Defaults to false, and true is a synonym for 20.

merge.suppressDest

By adding a glob that matches the names of integration branches to this multi-valued configuration variable, the default merge message computed for merges into these integration branches will omit "into <branch name>" from its title.

An element with an empty value can be used to clear the list of globs accumulated from previous configuration entries. When there is no *merge.suppressDest* variable defined, the default value of *master* is used for backward compatibility.

merge.renameLimit

The number of files to consider in the exhaustive portion of rename detection during a merge. If not specified, defaults to the value of diff.renameLimit. If neither merge.renameLimit nor diff.renameLimit are specified, currently defaults to 7000. This setting has no effect if rename detection is turned off.

merge.renames

Whether Git detects renames. If set to "false", rename detection is disabled. If set to "true", basic rename detection is enabled. Defaults to the value of diff.renames.

merge.directoryRenames

Whether Git detects directory renames, affecting what happens at merge time to new files added to a directory on one side of history when that directory was renamed on the other side of history. If merge.directoryRenames is set to "false", directory rename detection is disabled, meaning that such new files will be left behind in the old directory. If set to "true", directory rename detection is enabled, meaning that such new files will be moved into the new directory. If set to "conflict", a conflict will be reported for such paths. If merge.renames is false, merge.directoryRenames is ignored and treated as false. Defaults to "conflict".

merge.renormalize

Tell Git that canonical representation of files in the repository has changed over time (e.g. earlier commits record text files with CRLF line endings, but recent ones use LF line endings). In such a repository, Git can convert the data recorded in commits to a canonical form before performing a merge to reduce unnecessary conflicts. For more information, see section "Merging branches with differing checkin/checkout attributes" in Section G.4.2, “gitattributes(5)”.

merge.stat

Whether to print the diffstat between ORIG_HEAD and the merge result at the end of the merge. True by default.

merge.autoStash

When set to true, automatically create a temporary stash entry before the operation begins, and apply it after the operation ends. This means that you can run merge on a dirty worktree. However, use with care: the final stash application after a successful merge might result in non-trivial conflicts. This option can be overridden by the *--no-autostash* and *--autostash* options of Section G.3.86, “git-merge(1)”. Defaults to false.

merge.tool

Controls which merge tool is used by Section G.3.88, “git-mergetool(1)”. The list below shows the valid built-in values. Any other value is treated as a custom merge tool and requires that a corresponding mergetool.<tool>.cmd variable is defined.

merge.guitool

Controls which merge tool is used by Section G.3.88, “git-mergetool(1)” when the -g/--gui flag is specified. The list below shows the valid built-in values. Any other value is treated as a custom merge tool and requires that a corresponding mergetool.<guitool>.cmd variable is defined.

araxis

Use Araxis Merge (requires a graphical session)

bc

Use Beyond Compare (requires a graphical session)

bc3

Use Beyond Compare (requires a graphical session)

bc4

Use Beyond Compare (requires a graphical session)

codecompare

Use Code Compare (requires a graphical session)

deltawalker

Use DeltaWalker (requires a graphical session)

diffmerge

Use DiffMerge (requires a graphical session)

diffuse

Use Diffuse (requires a graphical session)

ecmerge

Use ECMerge (requires a graphical session)

emerge

Use Emacs' Emerge

examdiff

Use ExamDiff Pro (requires a graphical session)

guiffy

Use Guiffy's Diff Tool (requires a graphical session)

gvimdiff

Use gVim (requires a graphical session) with a custom layout (see *git help mergetool*'s *BACKEND SPECIFIC HINTS* section)

gvimdiff1

Use gVim (requires a graphical session) with a 2 panes layout (LOCAL and REMOTE)

gvimdiff2

Use gVim (requires a graphical session) with a 3 panes layout (LOCAL, MERGED and REMOTE)

gvimdiff3

Use gVim (requires a graphical session) where only the MERGED file is shown

kdiff3

Use KDiff3 (requires a graphical session)

meld

Use Meld (requires a graphical session) with optional *auto merge* (see *git help mergetool*'s *CONFIGURATION* section)

nvimdiff

Use Neovim with a custom layout (see *git help mergetool*'s *BACKEND SPECIFIC HINTS* section)

nvimdiff1

Use Neovim with a 2 panes layout (LOCAL and REMOTE)

nvimdiff2

Use Neovim with a 3 panes layout (LOCAL, MERGED and REMOTE)

nvimdiff3

Use Neovim where only the MERGED file is shown

opendiff

Use FileMerge (requires a graphical session)

p4merge

Use HelixCore P4Merge (requires a graphical session)

smerge

Use Sublime Merge (requires a graphical session)

tkdiff

Use TkDiff (requires a graphical session)

tortoisemerge

Use TortoiseMerge (requires a graphical session)

vimdiff

Use Vim with a custom layout (see *git help mergetool*'s *BACKEND SPECIFIC HINTS* section)

vimdiff1

Use Vim with a 2 panes layout (LOCAL and REMOTE)

vimdiff2

Use Vim with a 3 panes layout (LOCAL, MERGED and REMOTE)

vimdiff3

Use Vim where only the MERGED file is shown

winmerge

Use WinMerge (requires a graphical session)

xxdiff

Use xxdiff (requires a graphical session)

merge.verbosity

Controls the amount of output shown by the recursive merge strategy. Level 0 outputs nothing except a final error message if conflicts were detected. Level 1 outputs only conflicts, 2 outputs conflicts and file

changes. Level 5 and above outputs debugging information. The default is level 2. Can be overridden by the *GIT_MERGE_VERBOSITY* environment variable.

merge.<driver>.name

Defines a human-readable name for a custom low-level merge driver. See Section G.4.2, “gitattributes(5)” for details.

merge.<driver>.driver

Defines the command that implements a custom low-level merge driver. See Section G.4.2, “gitattributes(5)” for details.

merge.<driver>.recursive

Names a low-level merge driver to be used when performing an internal merge between common ancestors. See Section G.4.2, “gitattributes(5)” for details.

SEE ALSO

Section G.3.51, “git-fmt-merge-msg(1)”, Section G.3.102, “git-pull(1)”, Section G.4.2, “gitattributes(5)”, Section G.3.118, “git-reset(1)”, Section G.3.44, “git-diff(1)”, Section G.3.75, “git-ls-files(1)”, Section G.3.2, “git-add(1)”, Section G.3.123, “git-rm(1)”, Section G.3.88, “git-mergetool(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.87. git-mergetool--lib(1)

2024-07-02

Revision History

2024-07-02

NAME

git-mergetool--lib - Common Git merge tool shell scriptlets

SYNOPSIS

TOOL_MODE=(diff|merge) . "$(git --exec-path)/git-mergetool--lib"

DESCRIPTION

This is not a command the end user would want to run. Ever. This documentation is meant for people who are studying the Porcelain-ish scripts and/or are writing new ones.

The *git-mergetool--lib* scriptlet is designed to be sourced (using .) by other shell scripts to set up functions for working with Git merge tools.

Before sourcing *git-mergetool--lib*, your script must set *TOOL_MODE* to define the operation mode for the functions listed below. *diff* and *merge* are valid values.

FUNCTIONS

get_merge_tool

Returns a merge tool. The return code is 1 if we returned a guessed merge tool, else 0. *$GIT_MERGETOOL_GUI* may be set to *true* to search for the appropriate guitool.

get_merge_tool_cmd

Returns the custom command for a merge tool.

get_merge_tool_path

Returns the custom path for a merge tool.

initialize_merge_tool

Brings merge tool specific functions into scope so they can be used or overridden.

run_merge_tool

Launches a merge tool given the tool name and a true/false flag to indicate whether a merge base is present. *$MERGED*, *$LOCAL*, *$REMOTE*, and *$BASE* must be defined for use by the merge tool.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.88. git-mergetool(1)

2024-07-02
Revision History
2024-07-02

NAME

git-mergetool - Run merge conflict resolution tools to resolve merge conflicts

SYNOPSIS

git mergetool [--tool=<tool>] [-y | --[no-]prompt] [<file>…]

DESCRIPTION

Use *git mergetool* to run one of several merge utilities to resolve merge conflicts. It is typically run after *git merge*.

If one or more <file> parameters are given, the merge tool program will be run to resolve differences in each file (skipping those without conflicts). Specifying a directory will include all unresolved files in that path. If no <file> names are specified, *git mergetool* will run the merge tool program on every file with merge conflicts.

OPTIONS

-t <tool> , --tool=<tool>

Use the merge resolution program specified by <tool>. Valid values include emerge, gvimdiff, kdiff3, meld, vimdiff, and tortoisemerge. Run *git mergetool --tool-help* for the list of valid <tool> settings.

If a merge resolution program is not specified, *git mergetool* will use the configuration variable *merge.tool*. If the configuration variable *merge.tool* is not set, *git mergetool* will pick a suitable default.

You can explicitly provide a full path to the tool by setting the configuration variable *mergetool.<tool>.path*. For example, you can configure the absolute path to kdiff3 by setting *mergetool.kdiff3.path*. Otherwise, *git mergetool* assumes the tool is available in PATH.

Instead of running one of the known merge tool programs, *git mergetool* can be customized to run an alternative program by specifying the command line to invoke in a configuration variable *mergetool.<tool>.cmd*.

When *git mergetool* is invoked with this tool (either through the *-t* or *--tool* option or the *merge.tool* configuration variable), the configured command line will be invoked with *$BASE* set to the name of a temporary file containing the common base for the merge, if available; *$LOCAL* set to the name of a temporary file containing the contents of the file on the current branch; *$REMOTE* set to the name of a temporary file containing the contents of the file to be merged, and *$MERGED* set to the name of the file to which the merge tool should write the result of the merge resolution.

If the custom merge tool correctly indicates the success of a merge resolution with its exit code, then the configuration variable *mergetool.<tool>.trustExitCode* can be set to *true*. Otherwise, *git mergetool* will prompt the user to indicate the success of the resolution after the custom tool has exited.

--tool-help

Print a list of merge tools that may be used with *--tool*.

-y , --no-prompt

Don't prompt before each invocation of the merge resolution program. This is the default if the merge resolution program is explicitly specified with the *--tool* option or with the *merge.tool* configuration variable.

--prompt

Prompt before each invocation of the merge resolution program to give the user a chance to skip the path.

-g , --gui

When *git-mergetool* is invoked with the *-g* or *--gui* option, the default merge tool will be read from the configured *merge.guitool* variable instead of *merge.tool*. If *merge.guitool* is not set, we will fallback to the tool configured under *merge.tool*. This may be autoselected using the configuration variable *mergetool.guiDefault*.

--no-gui

This overrides a previous *-g* or *--gui* setting or *mergetool.guiDefault* configuration and reads the default merge tool from the configured *merge.tool* variable.

-O<orderfile>

Process files in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the *diff.orderFile* configuration variable (see Section G.3.29, “git-config(1)”). To cancel *diff.orderFile*, use *-O/dev/null*.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

mergetool.<tool>.path

Override the path for the given tool. This is useful in case your tool is not in the PATH.

mergetool.<tool>.cmd

Specify the command to invoke the specified merge tool. The specified command is evaluated in shell with the following variables available: *BASE* is the name of a temporary file containing the common base of the files to be merged, if available; *LOCAL* is the name of a temporary file containing the contents of the file on the current branch; *REMOTE* is the name of a temporary file containing the contents of the file from the branch being merged; *MERGED* contains the name of the file to which the merge tool should write the results of a successful merge.

mergetool.<tool>.hideResolved

Allows the user to override the global *mergetool.hideResolved* value for a specific tool. See *mergetool.hideResolved* for the full description.

mergetool.<tool>.trustExitCode

For a custom merge command, specify whether the exit code of the merge command can be used to determine whether the merge was successful. If this is not set to true then the merge target file timestamp is checked, and the merge is assumed to have been successful if the file has been updated; otherwise, the user is prompted to indicate the success of the merge.

mergetool.meld.hasOutput

Older versions of *meld* do not support the *--output* option. Git will attempt to detect whether *meld* supports *--output* by inspecting the output of *meld --help*. Configuring *mergetool.meld.hasOutput* will make Git skip these checks and use the configured value instead. Setting *mergetool.meld.hasOutput* to *true* tells Git to unconditionally use the *--output* option, and *false* avoids using *--output*.

mergetool.meld.useAutoMerge

When the *--auto-merge* is given, meld will merge all non-conflicting parts automatically, highlight the conflicting parts, and wait for user decision. Setting *mergetool.meld.useAutoMerge* to *true* tells Git to unconditionally use the *--auto-merge* option with *meld*. Setting this value to *auto* makes git detect whether *--auto-merge* is supported and will only use *--auto-merge* when available. A value of *false* avoids using *--auto-merge* altogether, and is the default value.

mergetool.<vimdiff variant>.layout

Configure the split window layout for vimdiff's *<variant>*, which is any of *vimdiff*, *nvimdiff*, *gvimdiff*. Upon launching *git mergetool* with *--tool=<variant>* (or without *--tool* if *merge.tool* is configured as *<variant>*), Git will consult *mergetool.<variant>.layout* to determine the tool's layout. If the variant-specific configuration is not available, *vimdiff*'s is used as fallback. If that too is not available, a default layout with 4 windows will be used. To configure the layout, see the *BACKEND SPECIFIC HINTS* section.

mergetool.hideResolved

During a merge, Git will automatically resolve as many conflicts as possible and write the *MERGED* file containing conflict markers around any conflicts that it cannot resolve; *LOCAL* and *REMOTE* normally represent the versions of the file from before Git's conflict resolution. This flag causes *LOCAL* and *REMOTE* to be overwritten so that only the unresolved conflicts are presented to the merge tool. Can be configured per-tool via the *mergetool.<tool>.hideResolved* configuration variable. Defaults to *false*.

mergetool.keepBackup

After performing a merge, the original file with conflict markers can be saved as a file with a *.orig* extension. If this variable is set to *false* then this file is not preserved. Defaults to *true* (i.e. keep the backup files).

mergetool.keepTemporaries

When invoking a custom merge tool, Git uses a set of temporary files to pass to the tool. If the tool returns an error and this variable is set to *true*, then these temporary files will be preserved; otherwise, they will be removed after the tool has exited. Defaults to *false*.

mergetool.writeToTemp

Git writes temporary *BASE*, *LOCAL*, and *REMOTE* versions of conflicting files in the worktree by default. Git will attempt to use a temporary directory for these files when set *true*. Defaults to *false*.

mergetool.prompt

Prompt before each invocation of the merge resolution program.

mergetool.guiDefault

Set *true* to use the *merge.guitool* by default (equivalent to specifying the *--gui* argument), or *auto* to select *merge.guitool* or *merge.tool* depending on the presence of a *DISPLAY* environment variable value. The default is *false*, where the *--gui* argument must be provided explicitly for the *merge.guitool* to be used.

TEMPORARY FILES

git mergetool creates **.orig* backup files while resolving merges. These are safe to remove once a file has been merged and its *git mergetool* session has completed.

Setting the *mergetool.keepBackup* configuration variable to *false* causes *git mergetool* to automatically remove the backup files as files are successfully merged.

BACKEND SPECIFIC HINTS

1. vimdiff

1.1. Description

When specifying *--tool=vimdiff* in *git mergetool* Git will open Vim with a 4 windows layout distributed in the following way:

```
-----------------------------------------
|              |            |              |
|   LOCAL      |   BASE     |   REMOTE     |
|              |            |              |
-----------------------------------------
|                                         |
|                 MERGED                  |
|                                         |
-----------------------------------------
```

LOCAL, *BASE* and *REMOTE* are read-only buffers showing the contents of the conflicting file in specific commits ("commit you are merging into", "common ancestor commit" and "commit you are merging from" respectively)

MERGED is a writable buffer where you have to resolve the conflicts (using the other read-only buffers as a reference). Once you are done, save and exit Vim as usual (*:wq*) or, if you want to abort, exit using *:cq*.

1.2. Layout configuration

You can change the windows layout used by Vim by setting configuration variable *mergetool.vimdiff.layout* which accepts a string where the following separators have special meaning:

- `+` is used to "open a new tab"
- `,` is used to "open a new vertical split"
- `/` is used to "open a new horizontal split"
- `@` is used to indicate the file containing the final version after solving the conflicts. If not present, *MERGED* will be used by default.

The precedence of the operators is as follows (you can use parentheses to change it):

```
`@` > `+` > `/` > `,`
```

Let's see some examples to understand how it works:

- *layout = "(LOCAL,BASE,REMOTE)/MERGED"*

  This is exactly the same as the default layout we have already seen.

  Note that `/` has precedence over `,` and thus the parenthesis are not needed in this case. The next layout definition is equivalent:

  ```
  layout = "LOCAL,BASE,REMOTE / MERGED"
  ```

- *layout = "LOCAL,MERGED,REMOTE"*

  If, for some reason, we are not interested in the *BASE* buffer.

```
-----------------------------------------
|              |            |            |
|              |            |            |
|   LOCAL      |   MERGED   |   REMOTE   |
|              |            |            |
|              |            |            |
-----------------------------------------
```

- *layout = "MERGED"*

  Only the *MERGED* buffer will be shown. Note, however, that all the other ones are still loaded in vim, and you can access them with the "buffers" command.

```
-----------------------------------------
|                                       |
|                                       |
|                MERGED                 |
|                                       |
|                                       |
-----------------------------------------
```

- *layout = "@LOCAL,REMOTE"*

  When *MERGED* is not present in the layout, you must "mark" one of the buffers with an asterisk. That will become the buffer you need to edit and save after resolving the conflicts.

```
-----------------------------------------
|                    |                  |
|                    |                  |
|                    |                  |
|      LOCAL         |    REMOTE        |
|                    |                  |
|                    |                  |
|                    |                  |
-----------------------------------------
```

- *layout = "LOCAL,BASE,REMOTE / MERGED + BASE,LOCAL + BASE,REMOTE"*

  Three tabs will open: the first one is a copy of the default layout, while the other two only show the differences between (*BASE* and *LOCAL*) and (*BASE* and *REMOTE*) respectively.

```
-----------------------------------------
| <TAB #1> |  TAB #2  |  TAB #3  |       |
-----------------------------------------
|              |            |            |
|   LOCAL      |   BASE     |   REMOTE   |
|              |            |            |
-----------------------------------------
|                                       |
|                MERGED                 |
|                                       |
-----------------------------------------

-----------------------------------------
|  TAB #1  | <TAB #2> |  TAB #3  |       |
-----------------------------------------
|                    |                  |
|                    |                  |
|                    |                  |
```

```
|      BASE           |     LOCAL           |
|                     |                     |
|                     |                     |
|                     |                     |
-----------------------------------------

-----------------------------------------
|  TAB #1  |  TAB #2  | <TAB #3> |        |
-----------------------------------------
|                     |                     |
|                     |                     |
|                     |                     |
|      BASE           |     REMOTE          |
|                     |                     |
|                     |                     |
|                     |                     |
-----------------------------------------
```

- *layout = "LOCAL,BASE,REMOTE / MERGED + BASE,LOCAL + BASE,REMOTE + (LOCAL/BASE/REMOTE),MERGED"*

  Same as the previous example, but adds a fourth tab with the same information as the first tab, with a different layout.

```
---------------------------------------------
|  TAB #1  |  TAB #2  |  TAB #3  | <TAB #4> |
---------------------------------------------
|       LOCAL         |                     |
|---------------------|                     |
|       BASE          |        MERGED       |
|---------------------|                     |
|       REMOTE        |                     |
---------------------------------------------
```

  Note how in the third tab definition we need to use parentheses to make , have precedence over /.

1.3. Variants

Instead of *--tool=vimdiff*, you can also use one of these other variants:

- *--tool=gvimdiff*, to open gVim instead of Vim.
- *--tool=nvimdiff*, to open Neovim instead of Vim.

When using these variants, in order to specify a custom layout you will have to set configuration variables *mergetool.gvimdiff.layout* and *mergetool.nvimdiff.layout* instead of *mergetool.vimdiff.layout* (though the latter will be used as fallback if the variant-specific one is not set).

In addition, for backwards compatibility with previous Git versions, you can also append *1*, *2* or *3* to either *vimdiff* or any of the variants (ex: *vimdiff3*, *nvimdiff1*, etc…) to use a predefined layout. In other words, using *--tool=[g,n,]vimdiffx* is the same as using *--tool=[g,n,]vimdiff* and setting configuration variable *mergetool.[g,n,]vimdiff.layout* to…

- *x=1*: *"@LOCAL, REMOTE"*
- *x=2*: *"LOCAL, MERGED, REMOTE"*
- *x=3*: *"MERGED"*

Example: using *--tool=gvimdiff2* will open *gvim* with three columns (LOCAL, MERGED and REMOTE).

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.89. git-mktag(1)

2024-07-02
Revision History
2024-07-02

NAME

git-mktag - Creates a tag object with extra validation

SYNOPSIS

git mktag

DESCRIPTION

Reads a tag's contents on standard input and creates a tag object. The output is the new tag's <object> identifier.

This command is mostly equivalent to Section G.3.62, “git-hash-object(1)” invoked with *-t tag -w --stdin*. I.e. both of these will create and write a tag found in *my-tag*:

```
git mktag <my-tag
git hash-object -t tag -w --stdin <my-tag
```

The difference is that mktag will die before writing the tag if the tag doesn't pass a Section G.3.56, “git-fsck(1)” check.

The "fsck" check done by mktag is stricter than what Section G.3.56, “git-fsck(1)” would run by default in that all *fsck.<msg-id>* messages are promoted from warnings to errors (so e.g. a missing "tagger" line is an error).

Extra headers in the object are also an error under mktag, but ignored by Section G.3.56, “git-fsck(1)”. This extra check can be turned off by setting the appropriate *fsck.<msg-id>* variable:

```
git -c fsck.extraHeaderEntry=ignore mktag <my-tag-with-headers
```

OPTIONS

--strict

By default mktag turns on the equivalent of Section G.3.56, “git-fsck(1)” *--strict* mode. Use *--no-strict* to disable it.

Tag Format

A tag signature file, to be fed to this command's standard input, has a very simple fixed format: four lines of

```
object <hash>
type <typename>
tag <tagname>
tagger <tagger>
```

followed by some *optional* free-form message (some tags created by older Git may not have a *tagger* line). The message, when it exists, is separated by a blank line from the header. The message part may contain a signature that Git itself doesn't care about, but that can be verified with gpg.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.90. git-mktree(1)

2024-07-02
Revision History
2024-07-02

NAME

git-mktree - Build a tree-object from ls-tree formatted text

SYNOPSIS

git mktree [-z] [--missing] [--batch]

DESCRIPTION

Reads standard input in non-recursive *ls-tree* output format, and creates a tree object. The order of the tree entries is normalized by mktree so pre-sorting the input is not required. The object name of the tree object built is written to the standard output.

OPTIONS

-z

Read the NUL-terminated *ls-tree -z* output instead.

--missing

Allow missing objects. The default behaviour (without this option) is to verify that each tree entry's hash identifies an existing object. This option has no effect on the treatment of gitlink entries (aka "submodules") which are always allowed to be missing.

--batch

Allow building of more than one tree object before exiting. Each tree is separated by a single blank line. The final newline is optional. Note - if the *-z* option is used, lines are terminated with NUL.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.91. git-mv(1)

2024-07-02
Revision History
2024-07-02

NAME

git-mv - Move or rename a file, a directory, or a symlink

SYNOPSIS

git mv [<options>] <source>… <destination>

DESCRIPTION

Move or rename a file, directory, or symlink.

```
git mv [-v] [-f] [-n] [-k] <source> <destination>
git mv [-v] [-f] [-n] [-k] <source> ... <destination-directory>
```

In the first form, it renames <source>, which must exist and be either a file, symlink or directory, to <destination>. In the second form, the last argument has to be an existing directory; the given sources will be moved into this directory.

The index is updated after successful completion, but the change must still be committed.

OPTIONS

-f , --force

Force renaming or moving of a file even if the <destination> exists.

-k

Skip move or rename actions which would lead to an error condition. An error happens when a source is neither existing nor controlled by Git, or when it would overwrite an existing file unless *-f* is given.

-n , --dry-run

Do nothing; only show what would happen

-v , --verbose

Report the names of files as they are moved.

SUBMODULES

Moving a submodule using a gitfile (which means they were cloned with a Git version 1.7.8 or newer) will update the gitfile and core.worktree setting to make the submodule work in the new location. It also will attempt to update the submodule.<name>.path setting in the Section G.4.10, “gitmodules(5)” file and stage that file (unless -n is used).

BUGS

Each time a superproject update moves a populated submodule (e.g. when switching between commits before and after the move) a stale submodule checkout will remain in the old location and an empty directory will appear in the new location. To populate the submodule again in the new location the user will have to run "git submodule update" afterwards. Removing the old directory is only safe when it uses a gitfile, as otherwise the history of the submodule will be deleted too. Both steps will be obsolete when recursive submodule update has been implemented.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.92. git-multi-pack-index(1)

2024-07-02

Revision History

2024-07-02

NAME

git-multi-pack-index - Write and verify multi-pack-indexes

SYNOPSIS

git multi-pack-index [--object-dir=<dir>] [--[no-]bitmap] <sub-command>

DESCRIPTION

Write or verify a multi-pack-index (MIDX) file.

OPTIONS

--object-dir=<dir>

Use given directory for the location of Git objects. We check *<dir>/packs/multi-pack-index* for the current MIDX file, and *<dir>/packs* for the pack-files to index.

<*dir*> must be an alternate of the current repository.

--[no-]progress

Turn progress on/off explicitly. If neither is specified, progress is shown if standard error is connected to a terminal. Supported by sub-commands *write*, *verify*, *expire*, and `repack.

The following subcommands are available:

write

Write a new MIDX file. The following options are available for the *write* sub-command:

--preferred-pack=<pack>

Optionally specify the tie-breaking pack used when multiple packs contain the same object. <*pack*> must contain at least one object. If not given, ties are broken in favor of the pack with the lowest mtime.

--[no-]bitmap

Control whether or not a multi-pack bitmap is written.

--stdin-packs

Write a multi-pack index containing only the set of line-delimited pack index basenames provided over stdin.

--refs-snapshot=<path>

With *--bitmap*, optionally specify a file which contains a "refs snapshot" taken prior to repacking.

A reference snapshot is composed of line-delimited OIDs corresponding to the reference tips, usually taken by *git repack* prior to generating a new pack. A line may optionally start with a + character to indicate that the reference which corresponds to that OID is "preferred" (see Section G.3.29, “git-config(1)”'s *pack.preferBitmapTips*.)

The file given at <*path*> is expected to be readable, and can contain duplicates. (If a given OID is given more than once, it is marked as preferred if at least one instance of it begins with the special + marker).

verify

Verify the contents of the MIDX file.

expire

Delete the pack-files that are tracked by the MIDX file, but have no objects referenced by the MIDX (with the exception of *.keep* packs and cruft packs). Rewrite the MIDX file afterward to remove all references to these pack-files.

repack

Create a new pack-file containing objects in small pack-files referenced by the multi-pack-index. If the size given by the *--batch-size=<size>* argument is zero, then create a pack containing all objects referenced by the multi-pack-index. For a non-zero batch size, Select the pack-files by examining packs from oldest-to-newest, computing the "expected size" by counting the number of objects in the pack referenced by the multi-pack-index, then divide by the total number of objects in the pack and multiply by the pack size. We select packs with expected size below the batch size until the set of packs have total expected size at least the batch size, or all pack-files are considered. If only one pack-file is selected, then do nothing. If a new pack-file is created, rewrite the multi-pack-index to reference the new pack-file. A later run of *git multi-pack-index expire* will delete the pack-files that were part of this batch.

If *repack.packKeptObjects* is *false*, then any pack-files with an associated *.keep* file will not be selected for the batch to repack.

EXAMPLES

- Write a MIDX file for the packfiles in the current *.git* directory.

```
$ git multi-pack-index write
```

- Write a MIDX file for the packfiles in the current *.git* directory with a corresponding bitmap.

```
$ git multi-pack-index write --preferred-pack=<pack> --bitmap
```

- Write a MIDX file for the packfiles in an alternate object store.

```
$ git multi-pack-index --object-dir <alt> write
```

- Verify the MIDX file for the packfiles in the current *.git* directory.

```
$ git multi-pack-index verify
```

SEE ALSO

See *The Multi-Pack-Index Design Document* [https://www.kernel.org/pub/software/scm/git/docs/technical/multi-pack-index.html] and Section G.5.5, "gitformat-pack(5)" for more information on the multi-pack-index feature and its file format.

GIT

Part of the Section G.3.1, "git(1)" suite

G.3.93. git-name-rev(1)

2024-07-02
Revision History
2024-07-02

NAME

git-name-rev - Find symbolic names for given revs

SYNOPSIS

git name-rev [--tags] [--refs=<pattern>]
(--all | --annotate-stdin | <commit-ish>…)

DESCRIPTION

Finds symbolic names suitable for human digestion for revisions given in any format parsable by *git rev-parse*.

OPTIONS

--tags

Do not use branch names, but only tags to name the commits

--refs=<pattern>

Only use refs whose names match a given shell pattern. The pattern can be a branch name, a tag name, or a fully qualified ref name. If given multiple times, use refs whose names match any of the given shell patterns. Use *--no-refs* to clear any previous ref patterns given.

--exclude=<pattern>

Do not use any ref whose name matches a given shell pattern. The pattern can be one of branch name, tag name or fully qualified ref name. If given multiple times, a ref will be excluded when it matches any of the

given patterns. When used together with --refs, a ref will be used as a match only when it matches at least one --refs pattern and does not match any --exclude patterns. Use *--no-exclude* to clear the list of exclude patterns.

--all

List all commits reachable from all refs

--annotate-stdin

Transform stdin by substituting all the 40-character SHA-1 hexes (say $hex) with "$hex ($rev_name)". When used with --name-only, substitute with "$rev_name", omitting $hex altogether. This option was called *--stdin* in older versions of Git.

For example:

```
$ cat sample.txt

An abbreviated revision 2ae0a9cb82 will not be substituted.
The full name after substitution is
 2ae0a9cb8298185a94e5998086f380a355dd8907,
while its tree object is 70d105cc79e63b81cfdcb08a15297c23e60b07ad

$ git name-rev --annotate-stdin <sample.txt

An abbreviated revision 2ae0a9cb82 will not be substituted.
The full name after substitution is
 2ae0a9cb8298185a94e5998086f380a355dd8907 (master),
while its tree object is 70d105cc79e63b81cfdcb08a15297c23e60b07ad

$ git name-rev --name-only --annotate-stdin <sample.txt

An abbreviated revision 2ae0a9cb82 will not be substituted.
The full name after substitution is master,
while its tree object is 70d105cc79e63b81cfdcb08a15297c23e60b07ad
```

--name-only

Instead of printing both the SHA-1 and the name, print only the name. If given with --tags the usual tag prefix of "tags/" is also omitted from the name, matching the output of *git-describe* more closely.

--no-undefined

Die with error code != 0 when a reference is undefined, instead of printing *undefined*.

--always

Show uniquely abbreviated commit object as fallback.

EXAMPLES

Given a commit, find out where it is relative to the local refs. Say somebody wrote you about that fantastic commit 33db5f4d9027a10e477ccf054b2c1ab94f74c85a. Of course, you look into the commit, but that only tells you what happened, but not the context.

Enter *git name-rev*:

```
% git name-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a
33db5f4d9027a10e477ccf054b2c1ab94f74c85a tags/v0.99~940
```

Now you are wiser, because you know that it happened 940 revisions before v0.99.

Another nice thing you can do is:

```
% git log | git name-rev --annotate-stdin
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.94. git-notes(1)

2024-07-02
Revision History
2024-07-02

NAME

git-notes - Add or inspect object notes

SYNOPSIS

git notes [list [<object>]]
git notes add [-f] [--allow-empty] [--[no-]separator | --separator=<paragraph-break>] [--[no-]stripspace] [-F <file> | -m <m
git notes copy [-f] (--stdin | <from-object> [<to-object>])
git notes append [--allow-empty] [--[no-]separator | --separator=<paragraph-break>] [--[no-]stripspace] [-F <file> | -m <m
git notes edit [--allow-empty] [<object>] [--[no-]stripspace]
git notes show [<object>]
git notes merge [-v | -q] [-s <strategy>] <notes-ref>
git notes merge --commit [-v | -q]
git notes merge --abort [-v | -q]
git notes remove [--ignore-missing] [--stdin] [<object>…]
git notes prune [-n] [-v]
git notes get-ref

DESCRIPTION

Adds, removes, or reads notes attached to objects, without touching the objects themselves.

By default, notes are saved to and read from *refs/notes/commits*, but this default can be overridden. See the OPTIONS, CONFIGURATION, and ENVIRONMENT sections below. If this ref does not exist, it will be quietly created when it is first needed to store a note.

A typical use of notes is to supplement a commit message without changing the commit itself. Notes can be shown by *git log* along with the original commit message. To distinguish these notes from the message stored in the commit object, the notes are indented like the message, after an unindented line saying "Notes (<refname>):" (or "Notes:" for *refs/notes/commits*).

Notes can also be added to patches prepared with *git format-patch* by using the *--notes* option. Such notes are added as a patch commentary after a three dash separator line.

To change which notes are shown by *git log*, see the "notes.displayRef" discussion in the section called “CONFIGURATION”.

See the "notes.rewrite.<command>" configuration for a way to carry notes across commands that rewrite commits.

SUBCOMMANDS

list

List the notes object for a given object. If no object is given, show a list of all note objects and the objects they annotate (in the format "<note-object> <annotated-object>"). This is the default subcommand if no subcommand is given.

add

Add notes for a given object (defaults to HEAD). Abort if the object already has notes (use *-f* to overwrite existing notes). However, if you're using *add* interactively (using an editor to supply the notes contents), then - instead of aborting - the existing notes will be opened in the editor (like the *edit* subcommand). If you specify multiple *-m* and *-F*, a blank line will be inserted between the messages. Use the *--separator* option to insert other delimiters.

copy

Copy the notes for the first object onto the second object (defaults to HEAD). Abort if the second object already has notes, or if the first object has none (use -f to overwrite existing notes to the second object). This subcommand is equivalent to: *git notes add [-f] -C $(git notes list <from-object>) <to-object>*

In *--stdin* mode, take lines in the format

```
<from-object> SP <to-object> [ SP <rest> ] LF
```

on standard input, and copy the notes from each <from-object> to its corresponding <to-object>. (The optional *<rest>* is ignored so that the command can read the input given to the *post-rewrite* hook.)

append

Append new message(s) given by *-m* or *-F* options to an existing note, or add them as a new note if one does not exist, for the object (defaults to HEAD). When appending to an existing note, a blank line is added before each new message as an inter-paragraph separator. The separator can be customized with the *--separator* option.

edit

Edit the notes for a given object (defaults to HEAD).

show

Show the notes for a given object (defaults to HEAD).

merge

Merge the given notes ref into the current notes ref. This will try to merge the changes made by the given notes ref (called "remote") since the merge-base (if any) into the current notes ref (called "local").

If conflicts arise and a strategy for automatically resolving conflicting notes (see the "NOTES MERGE STRATEGIES" section) is not given, the "manual" resolver is used. This resolver checks out the conflicting notes in a special worktree (*.git/NOTES_MERGE_WORKTREE*), and instructs the user to manually resolve the conflicts there. When done, the user can either finalize the merge with *git notes merge --commit*, or abort the merge with *git notes merge --abort*.

remove

Remove the notes for given objects (defaults to HEAD). When giving zero or one object from the command line, this is equivalent to specifying an empty note message to the *edit* subcommand.

prune

Remove all notes for non-existing/unreachable objects.

get-ref

Print the current notes ref. This provides an easy way to retrieve the current notes ref (e.g. from scripts).

OPTIONS

-f , --force

When adding notes to an object that already has notes, overwrite the existing notes (instead of aborting).

-m <msg> , --message=<msg>

Use the given note message (instead of prompting). If multiple *-m* options are given, their values are concatenated as separate paragraphs. Lines starting with # and empty lines other than a single line between paragraphs will be stripped out. If you wish to keep them verbatim, use *--no-stripspace*.

-F <file> , --file=<file>

Take the note message from the given file. Use - to read the note message from the standard input. Lines starting with # and empty lines other than a single line between paragraphs will be stripped out. If you wish to keep them verbatim, use *--no-stripspace*.

-C <object> , --reuse-message=<object>

Take the given blob object (for example, another note) as the note message. (Use *git notes copy <object>* instead to copy notes between objects.). By default, message will be copied verbatim, but if you wish to strip out the lines starting with # and empty lines other than a single line between paragraphs, use with`--stripspace` option.

-c <object> , --reedit-message=<object>

Like *-C*, but with *-c* the editor is invoked, so that the user can further edit the note message.

--allow-empty

Allow an empty note object to be stored. The default behavior is to automatically remove empty notes.

--[no-]separator, --separator=<paragraph-break>

Specify a string used as a custom inter-paragraph separator (a newline is added at the end as needed). If *--no-separator*, no separators will be added between paragraphs. Defaults to a blank line.

--[no-]stripspace

Strip leading and trailing whitespace from the note message. Also strip out empty lines other than a single line between paragraphs. Lines starting with # will be stripped out in non-editor cases like *-m*, *-F* and *-C*, but not in editor case like *git notes edit*, *-c*, etc.

--ref <ref>

Manipulate the notes tree in <ref>. This overrides *GIT_NOTES_REF* and the "core.notesRef" configuration. The ref specifies the full refname when it begins with *refs/notes/*; when it begins with *notes/*, *refs/* and otherwise *refs/notes/* is prefixed to form a full name of the ref.

--ignore-missing

Do not consider it an error to request removing notes from an object that does not have notes attached to it.

--stdin

Also read the object names to remove notes from the standard input (there is no reason you cannot combine this with object names from the command line).

-n , --dry-run

Do not remove anything; just report the object names whose notes would be removed.

-s <strategy> , --strategy=<strategy>

When merging notes, resolve notes conflicts using the given strategy. The following strategies are recognized: "manual" (default), "ours", "theirs", "union" and "cat_sort_uniq". This option overrides the "notes.mergeS-

trategy" configuration setting. See the "NOTES MERGE STRATEGIES" section below for more information on each notes merge strategy.

--commit

Finalize an in-progress *git notes merge*. Use this option when you have resolved the conflicts that *git notes merge* stored in .git/NOTES_MERGE_WORKTREE. This amends the partial merge commit created by *git notes merge* (stored in .git/NOTES_MERGE_PARTIAL) by adding the notes in .git/NOTES_MERGE_WORKTREE. The notes ref stored in the .git/NOTES_MERGE_REF symref is updated to the resulting commit.

--abort

Abort/reset an in-progress *git notes merge*, i.e. a notes merge with conflicts. This simply removes all files related to the notes merge.

-q , --quiet

When merging notes, operate quietly.

-v , --verbose

When merging notes, be more verbose. When pruning notes, report all object names whose notes are removed.

DISCUSSION

Commit notes are blobs containing extra information about an object (usually information to supplement a commit's message). These blobs are taken from notes refs. A notes ref is usually a branch which contains "files" whose paths are the object names for the objects they describe, with some directory separators included for performance reasons [1].

Every notes change creates a new commit at the specified notes ref. You can therefore inspect the history of the notes by invoking, e.g., *git log -p notes/commits*. Currently the commit message only records which operation triggered the update, and the commit authorship is determined according to the usual rules (see Section G.3.28, “git-commit(1)”). These details may change in the future.

It is also permitted for a notes ref to point directly to a tree object, in which case the history of the notes can be read with *git log -p -g <refname>*.

NOTES MERGE STRATEGIES

The default notes merge strategy is "manual", which checks out conflicting notes in a special work tree for resolving notes conflicts (*.git/NOTES_MERGE_WORKTREE*), and instructs the user to resolve the conflicts in that work tree. When done, the user can either finalize the merge with *git notes merge --commit*, or abort the merge with *git notes merge --abort*.

Users may select an automated merge strategy from among the following using either -s/--strategy option or configuring notes.mergeStrategy accordingly:

"ours" automatically resolves conflicting notes in favor of the local version (i.e. the current notes ref).

"theirs" automatically resolves notes conflicts in favor of the remote version (i.e. the given notes ref being merged into the current notes ref).

"union" automatically resolves notes conflicts by concatenating the local and remote versions.

"cat_sort_uniq" is similar to "union", but in addition to concatenating the local and remote versions, this strategy also sorts the resulting lines, and removes duplicate lines from the result. This is equivalent to applying the "cat |

[1] Permitted pathnames have the form *bf/fe/30/…/680d5a…*: a sequence of directory names of two hexadecimal digits each followed by a filename with the rest of the object ID.

sort | uniq" shell pipeline to the local and remote versions. This strategy is useful if the notes follow a line-based format where one wants to avoid duplicated lines in the merge result. Note that if either the local or remote version contain duplicate lines prior to the merge, these will also be removed by this notes merge strategy.

EXAMPLES

You can use notes to add annotations with information that was not available at the time a commit was written.

```
$ git notes add -m 'Tested-by: Johannes Sixt <j6t@kdbg.org>' 72a144e2
$ git show -s 72a144e
[...]
    Signed-off-by: Junio C Hamano <gitster@pobox.com>

Notes:
    Tested-by: Johannes Sixt <j6t@kdbg.org>
```

In principle, a note is a regular Git blob, and any kind of (non-)format is accepted. You can binary-safely create notes from arbitrary files using *git hash-object*:

```
$ cc *.c
$ blob=$(git hash-object -w a.out)
$ git notes --ref=built add --allow-empty -C "$blob" HEAD
```

(You cannot simply use *git notes --ref=built add -F a.out HEAD* because that is not binary-safe.) Of course, it doesn't make much sense to display non-text-format notes with *git log*, so if you use such notes, you'll probably need to write some special-purpose tools to do something useful with them.

CONFIGURATION

core.notesRef

Notes ref to read and manipulate instead of *refs/notes/commits*. Must be an unabbreviated ref name. This setting can be overridden through the environment and command line.

Everything above this line in this section isn't included from the Section G.3.29, “git-config(1)” documentation. The content that follows is the same as what's found there:

notes.mergeStrategy

Which merge strategy to choose by default when resolving notes conflicts. Must be one of *manual*, *ours*, *theirs*, *union*, or *cat_sort_uniq*. Defaults to *manual*. See the "NOTES MERGE STRATEGIES" section of Section G.3.94, “git-notes(1)” for more information on each strategy.

This setting can be overridden by passing the *--strategy* option to Section G.3.94, “git-notes(1)”.

notes.<name>.mergeStrategy

Which merge strategy to choose when doing a notes merge into refs/notes/<name>. This overrides the more general "notes.mergeStrategy". See the "NOTES MERGE STRATEGIES" section in Section G.3.94, “git-notes(1)” for more information on the available strategies.

notes.displayRef

Which ref (or refs, if a glob or specified more than once), in addition to the default set by *core.notesRef* or *GIT_NOTES_REF*, to read notes from when showing commit messages with the *git log* family of commands.

This setting can be overridden with the *GIT_NOTES_DISPLAY_REF* environment variable, which must be a colon separated list of refs or globs.

A warning will be issued for refs that do not exist, but a glob that does not match any refs is silently ignored.

This setting can be disabled by the *--no-notes* option to the *git log* family of commands, or by the *--notes=<ref>* option accepted by those commands.

The effective value of "core.notesRef" (possibly overridden by GIT_NOTES_REF) is also implicitly added to the list of refs to be displayed.

notes.rewrite.<command>

When rewriting commits with <command> (currently *amend* or *rebase*), if this variable is *false*, git will not copy notes from the original to the rewritten commit. Defaults to *true*. See also "*notes.rewriteRef*" below.

This setting can be overridden with the *GIT_NOTES_REWRITE_REF* environment variable, which must be a colon separated list of refs or globs.

notes.rewriteMode

When copying notes during a rewrite (see the "notes.rewrite.<command>" option), determines what to do if the target commit already has a note. Must be one of *overwrite*, *concatenate*, *cat_sort_uniq*, or *ignore*. Defaults to *concatenate*.

This setting can be overridden with the *GIT_NOTES_REWRITE_MODE* environment variable.

notes.rewriteRef

When copying notes during a rewrite, specifies the (fully qualified) ref whose notes should be copied. May be a glob, in which case notes in all matching refs will be copied. You may also specify this configuration several times.

Does not have a default value; you must configure this variable to enable note rewriting. Set it to *refs/notes/commits* to enable rewriting for the default commit notes.

Can be overridden with the *GIT_NOTES_REWRITE_REF* environment variable. See *notes.rewrite.<command>* above for a further description of its format.

ENVIRONMENT

GIT_NOTES_REF

Which ref to manipulate notes from, instead of *refs/notes/commits*. This overrides the *core.notesRef* setting.

GIT_NOTES_DISPLAY_REF

Colon-delimited list of refs or globs indicating which refs, in addition to the default from *core.notesRef* or *GIT_NOTES_REF*, to read notes from when showing commit messages. This overrides the *notes.displayRef* setting.

A warning will be issued for refs that do not exist, but a glob that does not match any refs is silently ignored.

GIT_NOTES_REWRITE_MODE

When copying notes during a rewrite, what to do if the target commit already has a note. Must be one of *overwrite*, *concatenate*, *cat_sort_uniq*, or *ignore*. This overrides the *core.rewriteMode* setting.

GIT_NOTES_REWRITE_REF

When rewriting commits, which notes to copy from the original to the rewritten commit. Must be a colon-delimited list of refs or globs.

If not set in the environment, the list of notes to copy depends on the *notes.rewrite.<command>* and *notes.rewriteRef* settings.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.95. git-p4(1)

2024-07-02
Revision History
2024-07-02

NAME

git-p4 - Import from and submit to Perforce repositories

SYNOPSIS

git p4 clone [<sync-options>] [<clone-options>] <p4-depot-path>…
git p4 sync [<sync-options>] [<p4-depot-path>…]
git p4 rebase
git p4 submit [<submit-options>] [<master-branch-name>]

DESCRIPTION

This command provides a way to interact with p4 repositories using Git.

Create a new Git repository from an existing p4 repository using *git p4 clone*, giving it one or more p4 depot paths. Incorporate new commits from p4 changes with *git p4 sync*. The *sync* command is also used to include new branches from other p4 depot paths. Submit Git changes back to p4 using *git p4 submit*. The command *git p4 rebase* does a sync plus rebases the current branch onto the updated p4 remote branch.

EXAMPLES

- Clone a repository:

```
$ git p4 clone //depot/path/project
```

- Do some work in the newly created Git repository:

```
$ cd project
$ vi foo.h
$ git commit -a -m "edited foo.h"
```

- Update the Git repository with recent changes from p4, rebasing your work on top:

```
$ git p4 rebase
```

- Submit your commits back to p4:

```
$ git p4 submit
```

COMMANDS

1. Clone

Generally, *git p4 clone* is used to create a new Git directory from an existing p4 repository:

```
$ git p4 clone //depot/path/project
```

This:

1. Creates an empty Git repository in a subdirectory called *project*.

2. Imports the full contents of the head revision from the given p4 depot path into a single commit in the Git branch *refs/remotes/p4/master*.

3. Creates a local branch, *master* from this remote and checks it out.

To reproduce the entire p4 history in Git, use the *@all* modifier on the depot path:

```
$ git p4 clone //depot/path/project@all
```

2. Sync

As development continues in the p4 repository, those changes can be included in the Git repository using:

```
$ git p4 sync
```

This command finds new changes in p4 and imports them as Git commits.

P4 repositories can be added to an existing Git repository using *git p4 sync* too:

```
$ mkdir repo-git
$ cd repo-git
$ git init
$ git p4 sync //path/in/your/perforce/depot
```

This imports the specified depot into *refs/remotes/p4/master* in an existing Git repository. The *--branch* option can be used to specify a different branch to be used for the p4 content.

If a Git repository includes branches *refs/remotes/origin/p4*, these will be fetched and consulted first during a *git p4 sync*. Since importing directly from p4 is considerably slower than pulling changes from a Git remote, this can be useful in a multi-developer environment.

If there are multiple branches, doing *git p4 sync* will automatically use the "BRANCH DETECTION" algorithm to try to partition new changes into the right branch. This can be overridden with the *--branch* option to specify just a single branch to update.

3. Rebase

A common working pattern is to fetch the latest changes from the p4 depot and merge them with local uncommitted changes. Often, the p4 repository is the ultimate location for all code, thus a rebase workflow makes sense. This command does *git p4 sync* followed by *git rebase* to move local commits on top of updated p4 changes.

```
$ git p4 rebase
```

4. Submit

Submitting changes from a Git repository back to the p4 repository requires a separate p4 client workspace. This should be specified using the *P4CLIENT* environment variable or the Git configuration variable *git-p4.client*. The p4 client must exist, but the client root will be created and populated if it does not already exist.

To submit all changes that are in the current Git branch but not in the *p4/master* branch, use:

```
$ git p4 submit
```

To specify a branch other than the current one, use:

```
$ git p4 submit topicbranch
```

To specify a single commit or a range of commits, use:

```
$ git p4 submit --commit <sha1>
```

```
$ git p4 submit --commit <sha1..sha1>
```

The upstream reference is generally *refs/remotes/p4/master*, but can be overridden using the *--origin=* command-line option.

The p4 changes will be created as the user invoking *git p4 submit*. The *--preserve-user* option will cause ownership to be modified according to the author of the Git commit. This option requires admin privileges in p4, which can be granted using *p4 protect*.

To shelve changes instead of submitting, use *--shelve* and *--update-shelve*:

```
$ git p4 submit --shelve
$ git p4 submit --update-shelve 1234 --update-shelve 2345
```

5. Unshelve

Unshelving will take a shelved P4 changelist, and produce the equivalent git commit in the branch refs/remotes/p4-unshelved/<changelist>.

The git commit is created relative to the current origin revision (HEAD by default). A parent commit is created based on the origin, and then the unshelve commit is created based on that.

The origin revision can be changed with the "--origin" option.

If the target branch in refs/remotes/p4-unshelved already exists, the old one will be renamed.

```
$ git p4 sync
$ git p4 unshelve 12345
$ git show p4-unshelved/12345
<submit more changes via p4 to the same files>
$ git p4 unshelve 12345
<refuses to unshelve until git is in sync with p4 again>
```

OPTIONS

1. General options

All commands except clone accept these options.

--git-dir <dir>

Set the *GIT_DIR* environment variable. See Section G.3.1, “git(1)”.

-v , --verbose

Provide more progress information.

2. Sync options

These options can be used in the initial *clone* as well as in subsequent *sync* operations.

--branch <ref>

Import changes into <ref> instead of refs/remotes/p4/master. If <ref> starts with refs/, it is used as is. Otherwise, if it does not start with p4/, that prefix is added.

By default a <ref> not starting with refs/ is treated as the name of a remote-tracking branch (under refs/remotes/). This behavior can be modified using the --import-local option.

The default <ref> is "master".

This example imports a new remote "p4/proj2" into an existing Git repository:

```
$ git init
$ git p4 sync --branch=refs/remotes/p4/proj2 //depot/proj2
```

--detect-branches

Use the branch detection algorithm to find new paths in p4. It is documented below in "BRANCH DETECTION".

--changesfile <file>

Import exactly the p4 change numbers listed in *file*, one per line. Normally, *git p4* inspects the current p4 repository state and detects the changes it should import.

--silent

Do not print any progress information.

--detect-labels

Query p4 for labels associated with the depot paths, and add them as tags in Git. Limited usefulness as only imports labels associated with new changelists. Deprecated.

--import-labels

Import labels from p4 into Git.

--import-local

By default, p4 branches are stored in *refs/remotes/p4/*, where they will be treated as remote-tracking branches by Section G.3.10, “git-branch(1)” and other commands. This option instead puts p4 branches in *refs/heads/p4/*. Note that future sync operations must specify *--import-local* as well so that they can find the p4 branches in refs/heads.

--max-changes <n>

Import at most *n* changes, rather than the entire range of changes included in the given revision specifier. A typical usage would be use *@all* as the revision specifier, but then to use *--max-changes 1000* to import only the last 1000 revisions rather than the entire revision history.

--changes-block-size <n>

The internal block size to use when converting a revision specifier such as *@all* into a list of specific change numbers. Instead of using a single call to *p4 changes* to find the full list of changes for the conversion, there are a sequence of calls to *p4 changes -m*, each of which requests one block of changes of the given size. The default block size is 500, which should usually be suitable.

--keep-path

The mapping of file names from the p4 depot path to Git, by default, involves removing the entire depot path. With this option, the full p4 depot path is retained in Git. For example, path *//depot/main/foo/bar.c*, when imported from *//depot/main/*, becomes *foo/bar.c*. With *--keep-path*, the Git path is instead *depot/main/foo/bar.c*.

--use-client-spec

Use a client spec to find the list of interesting files in p4. See the "CLIENT SPEC" section below.

-/ <path>

Exclude selected depot paths when cloning or syncing.

3. Clone options

These options can be used in an initial *clone*, along with the *sync* options described above.

--destination <directory>

Where to create the Git repository. If not provided, the last component in the p4 depot path is used to create a new directory.

--bare

Perform a bare clone. See Section G.3.24, “git-clone(1)”.

4. Submit options

These options can be used to modify *git p4 submit* behavior.

--origin <commit>

Upstream location from which commits are identified to submit to p4. By default, this is the most recent p4 commit reachable from *HEAD*.

-M

Detect renames. See Section G.3.44, “git-diff(1)”. Renames will be represented in p4 using explicit *move* operations. There is no corresponding option to detect copies, but there are variables for both moves and copies.

--preserve-user

Re-author p4 changes before submitting to p4. This option requires p4 admin privileges.

--export-labels

Export tags from Git as p4 labels. Tags found in Git are applied to the perforce working directory.

-n , --dry-run

Show just what commits would be submitted to p4; do not change state in Git or p4.

--prepare-p4-only

Apply a commit to the p4 workspace, opening, adding and deleting files in p4 as for a normal submit operation. Do not issue the final "p4 submit", but instead print a message about how to submit manually or revert. This option always stops after the first (oldest) commit. Git tags are not exported to p4.

--shelve

Instead of submitting create a series of shelved changelists. After creating each shelve, the relevant files are reverted/deleted. If you have multiple commits pending multiple shelves will be created.

--update-shelve CHANGELIST

Update an existing shelved changelist with this commit. Implies --shelve. Repeat for multiple shelved changelists.

--conflict=(ask|skip|quit)

Conflicts can occur when applying a commit to p4. When this happens, the default behavior ("ask") is to prompt whether to skip this commit and continue, or quit. This option can be used to bypass the prompt, causing conflicting commits to be automatically skipped, or to quit trying to apply commits, without prompting.

--branch <branch>

After submitting, sync this named branch instead of the default p4/master. See the "Sync options" section above for more information.

--commit (<sha1>|<sha1>..<sha1>)

Submit only the specified commit or range of commits, instead of the full list of changes that are in the current Git branch.

--disable-rebase

Disable the automatic rebase after all commits have been successfully submitted. Can also be set with git-p4.disableRebase.

--disable-p4sync

Disable the automatic sync of p4/master from Perforce after commits have been submitted. Implies --disable-rebase. Can also be set with git-p4.disableP4Sync. Sync with origin/master still goes ahead if possible.

Hooks for submit

1. p4-pre-submit

The *p4-pre-submit* hook is executed if it exists and is executable. The hook takes no parameters and nothing from standard input. Exiting with non-zero status from this script prevents *git-p4 submit* from launching. It can be bypassed with the *--no-verify* command line option.

One usage scenario is to run unit tests in the hook.

2. p4-prepare-changelist

The *p4-prepare-changelist* hook is executed right after preparing the default changelist message and before the editor is started. It takes one parameter, the name of the file that contains the changelist text. Exiting with a non-zero status from the script will abort the process.

The purpose of the hook is to edit the message file in place, and it is not suppressed by the *--no-verify* option. This hook is called even if *--prepare-p4-only* is set.

3. p4-changelist

The *p4-changelist* hook is executed after the changelist message has been edited by the user. It can be bypassed with the *--no-verify* option. It takes a single parameter, the name of the file that holds the proposed changelist text. Exiting with a non-zero status causes the command to abort.

The hook is allowed to edit the changelist file and can be used to normalize the text into some project standard format. It can also be used to refuse the Submit after inspect the message file.

4. p4-post-changelist

The *p4-post-changelist* hook is invoked after the submit has successfully occurred in P4. It takes no parameters and is meant primarily for notification and cannot affect the outcome of the git p4 submit action.

5. Rebase options

These options can be used to modify *git p4 rebase* behavior.

--import-labels

Import p4 labels.

6. Unshelve options

--origin

Sets the git refspec against which the shelved P4 changelist is compared. Defaults to p4/master.

DEPOT PATH SYNTAX

The p4 depot path argument to *git p4 sync* and *git p4 clone* can be one or more space-separated p4 depot paths, with an optional p4 revision specifier on the end:

"//depot/my/project"

Import one commit with all files in the *#head* change under that tree.

"//depot/my/project@all"

Import one commit for each change in the history of that depot path.

"//depot/my/project@1,6"

Import only changes 1 through 6.

"//depot/proj1@all //depot/proj2@all"

Import all changes from both named depot paths into a single repository. Only files below these directories are included. There is not a subdirectory in Git for each "proj1" and "proj2". You must use the *--destination* option when specifying more than one depot path. The revision specifier must be specified identically on each depot path. If there are files in the depot paths with the same name, the path with the most recently updated version of the file is the one that appears in Git.

See *p4 help revisions* for the full syntax of p4 revision specifiers.

CLIENT SPEC

The p4 client specification is maintained with the *p4 client* command and contains among other fields, a View that specifies how the depot is mapped into the client repository. The *clone* and *sync* commands can consult the client spec when given the *--use-client-spec* option or when the useClientSpec variable is true. After *git p4 clone*, the useClientSpec variable is automatically set in the repository configuration file. This allows future *git p4 submit* commands to work properly; the submit command looks only at the variable and does not have a command-line option.

The full syntax for a p4 view is documented in *p4 help views*. *git p4* knows only a subset of the view syntax. It understands multi-line mappings, overlays with +, exclusions with - and double-quotes around whitespace. Of the possible wildcards, *git p4* only handles …, and only when it is at the end of the path. *git p4* will complain if it encounters an unhandled wildcard.

Bugs in the implementation of overlap mappings exist. If multiple depot paths map through overlays to the same location in the repository, *git p4* can choose the wrong one. This is hard to solve without dedicating a client spec just for *git p4*.

The name of the client can be given to *git p4* in multiple ways. The variable *git-p4.client* takes precedence if it exists. Otherwise, normal p4 mechanisms of determining the client are used: environment variable *P4CLIENT*, a file referenced by *P4CONFIG*, or the local host name.

BRANCH DETECTION

P4 does not have the same concept of a branch as Git. Instead, p4 organizes its content as a directory tree, where by convention different logical branches are in different locations in the tree. The *p4 branch* command is used to

maintain mappings between different areas in the tree, and indicate related content. *git p4* can use these mappings to determine branch relationships.

If you have a repository where all the branches of interest exist as subdirectories of a single depot path, you can use *--detect-branches* when cloning or syncing to have *git p4* automatically find subdirectories in p4, and to generate these as branches in Git.

For example, if the P4 repository structure is:

```
//depot/main/...
//depot/branch1/...
```

And "p4 branch -o branch1" shows a View line that looks like:

```
//depot/main/... //depot/branch1/...
```

Then this *git p4 clone* command:

```
git p4 clone --detect-branches //depot@all
```

produces a separate branch in *refs/remotes/p4/* for //depot/main, called *master*, and one for //depot/branch1 called *depot/branch1*.

However, it is not necessary to create branches in p4 to be able to use them like branches. Because it is difficult to infer branch relationships automatically, a Git configuration setting *git-p4.branchList* can be used to explicitly identify branch relationships. It is a list of "source:destination" pairs, like a simple p4 branch specification, where the "source" and "destination" are the path elements in the p4 repository. The example above relied on the presence of the p4 branch. Without p4 branches, the same result will occur with:

```
git init depot
cd depot
git config git-p4.branchList main:branch1
git p4 clone --detect-branches //depot@all .
```

PERFORMANCE

The fast-import mechanism used by *git p4* creates one pack file for each invocation of *git p4 sync*. Normally, Git garbage compression (Section G.3.58, “git-gc(1)”) automatically compresses these to fewer pack files, but explicit invocation of *git repack -adf* may improve performance.

CONFIGURATION VARIABLES

The following config settings can be used to modify *git p4* behavior. They all are in the *git-p4* section.

1. General variables

git-p4.user

User specified as an option to all p4 commands, with *-u <user>*. The environment variable *P4USER* can be used instead.

git-p4.password

Password specified as an option to all p4 commands, with *-P <password>*. The environment variable *P4PASS* can be used instead.

git-p4.port

Port specified as an option to all p4 commands, with *-p <port>*. The environment variable *P4PORT* can be used instead.

git-p4.host

Host specified as an option to all p4 commands, with *-h <host>*. The environment variable *P4HOST* can be used instead.

git-p4.client

Client specified as an option to all p4 commands, with *-c <client>*, including the client spec.

git-p4.retries

Specifies the number of times to retry a p4 command (notably, *p4 sync*) if the network times out. The default value is 3. Set the value to 0 to disable retries or if your p4 version does not support retries (pre 2012.2).

2. Clone and sync variables

git-p4.syncFromOrigin

Because importing commits from other Git repositories is much faster than importing them from p4, a mechanism exists to find p4 changes first in Git remotes. If branches exist under *refs/remote/origin/p4*, those will be fetched and used when syncing from p4. This variable can be set to *false* to disable this behavior.

git-p4.branchUser

One phase in branch detection involves looking at p4 branches to find new ones to import. By default, all branches are inspected. This option limits the search to just those owned by the single user named in the variable.

git-p4.branchList

List of branches to be imported when branch detection is enabled. Each entry should be a pair of branch names separated by a colon (:). This example declares that both branchA and branchB were created from main:

```
git config       git-p4.branchList main:branchA
git config --add git-p4.branchList main:branchB
```

git-p4.ignoredP4Labels

List of p4 labels to ignore. This is built automatically as unimportable labels are discovered.

git-p4.importLabels

Import p4 labels into git, as per --import-labels.

git-p4.labelImportRegexp

Only p4 labels matching this regular expression will be imported. The default value is *[a-zA-Z0-9_\-.]+$*.

git-p4.useClientSpec

Specify that the p4 client spec should be used to identify p4 depot paths of interest. This is equivalent to specifying the option *--use-client-spec*. See the "CLIENT SPEC" section above. This variable is a boolean, not the name of a p4 client.

git-p4.pathEncoding

Perforce keeps the encoding of a path as given by the originating OS. Git expects paths encoded as UTF-8. Use this config to tell git-p4 what encoding Perforce had used for the paths. This encoding is used to transcode the paths to UTF-8. As an example, Perforce on Windows often uses "cp1252" to encode path names. If this option is passed into a p4 clone request, it is persisted in the resulting new git repo.

git-p4.metadataDecodingStrategy

Perforce keeps the encoding of a changelist descriptions and user full names as stored by the client on a given OS. The p4v client uses the OS-local encoding, and so different users can end up storing different changelist descriptions or user full names in different encodings, in the same depot. Git tolerates inconsistent/incorrect encodings in commit messages and author names, but expects them to be specified in utf-8. git-p4 can use three different decoding strategies in handling the encoding uncertainty in Perforce: *passthrough* simply passes the original bytes through from Perforce to git, creating usable but incorrectly-encoded data when the Perforce data is encoded as anything other than utf-8. *strict* expects the Perforce data to be encoded as utf-8, and fails to import when this is not true. *fallback* attempts to interpret the data as utf-8, and otherwise falls back to using a secondary encoding - by default the common windows encoding *cp-1252* - with upper-range bytes escaped if decoding with the fallback encoding also fails. Under python2 the default strategy is *passthrough* for historical reasons, and under python3 the default is *fallback*. When *strict* is selected and decoding fails, the error message will propose changing this config parameter as a workaround. If this option is passed into a p4 clone request, it is persisted into the resulting new git repo.

git-p4.metadataFallbackEncoding

Specify the fallback encoding to use when decoding Perforce author names and changelists descriptions using the *fallback* strategy (see git-p4.metadataDecodingStrategy). The fallback encoding will only be used when decoding as utf-8 fails. This option defaults to cp1252, a common windows encoding. If this option is passed into a p4 clone request, it is persisted into the resulting new git repo.

git-p4.largeFileSystem

Specify the system that is used for large (binary) files. Please note that large file systems do not support the *git p4 submit* command. Only Git LFS is implemented right now (see https://git-lfs.github.com/ for more information). Download and install the Git LFS command line extension to use this option and configure it like this:

```
git config       git-p4.largeFileSystem GitLFS
```

git-p4.largeFileExtensions

All files matching a file extension in the list will be processed by the large file system. Do not prefix the extensions with ..

git-p4.largeFileThreshold

All files with an uncompressed size exceeding the threshold will be processed by the large file system. By default the threshold is defined in bytes. Add the suffix k, m, or g to change the unit.

git-p4.largeFileCompressedThreshold

All files with a compressed size exceeding the threshold will be processed by the large file system. This option might slow down your clone/sync process. By default the threshold is defined in bytes. Add the suffix k, m, or g to change the unit.

git-p4.largeFilePush

Boolean variable which defines if large files are automatically pushed to a server.

git-p4.keepEmptyCommits

A changelist that contains only excluded files will be imported as an empty commit if this boolean option is set to true.

git-p4.mapUser

Map a P4 user to a name and email address in Git. Use a string with the following format to create a mapping:

```
git config --add git-p4.mapUser "p4user = First Last <mail@address.com>"
```

A mapping will override any user information from P4. Mappings for multiple P4 user can be defined.

3. Submit variables

git-p4.detectRenames

Detect renames. See Section G.3.44, “git-diff(1)”. This can be true, false, or a score as expected by *git diff -M*.

git-p4.detectCopies

Detect copies. See Section G.3.44, “git-diff(1)”. This can be true, false, or a score as expected by *git diff -C*.

git-p4.detectCopiesHarder

Detect copies harder. See Section G.3.44, “git-diff(1)”. A boolean.

git-p4.preserveUser

On submit, re-author changes to reflect the Git author, regardless of who invokes *git p4 submit*.

git-p4.allowMissingP4Users

When *preserveUser* is true, *git p4* normally dies if it cannot find an author in the p4 user map. This setting submits the change regardless.

git-p4.skipSubmitEdit

The submit process invokes the editor before each p4 change is submitted. If this setting is true, though, the editing step is skipped.

git-p4.skipSubmitEditCheck

After editing the p4 change message, *git p4* makes sure that the description really was changed by looking at the file modification time. This option disables that test.

git-p4.allowSubmit

By default, any branch can be used as the source for a *git p4 submit* operation. This configuration variable, if set, permits only the named branches to be used as submit sources. Branch names must be the short names (no "refs/heads/"), and should be separated by commas (","), with no spaces.

git-p4.skipUserNameCheck

If the user running *git p4 submit* does not exist in the p4 user map, *git p4* exits. This option can be used to force submission regardless.

git-p4.attemptRCSCleanup

If enabled, *git p4 submit* will attempt to cleanup RCS keywords ($Header$, etc). These would otherwise cause merge conflicts and prevent the submit going ahead. This option should be considered experimental at present.

git-p4.exportLabels

Export Git tags to p4 labels, as per --export-labels.

git-p4.labelExportRegexp

Only p4 labels matching this regular expression will be exported. The default value is *[a-zA-Z0-9_\-.]+$*.

git-p4.conflict

Specify submit behavior when a conflict with p4 is found, as per --conflict. The default behavior is *ask*.

git-p4.disableRebase

Do not rebase the tree against p4/master following a submit.

git-p4.disableP4Sync

Do not sync p4/master with Perforce following a submit. Implies git-p4.disableRebase.

IMPLEMENTATION DETAILS

- Changesets from p4 are imported using Git fast-import.
- Cloning or syncing does not require a p4 client; file contents are collected using *p4 print*.
- Submitting requires a p4 client, which is not in the same location as the Git repository. Patches are applied, one at a time, to this p4 client and submitted from there.
- Each commit imported by *git p4* has a line at the end of the log message indicating the p4 depot location and change number. This line is used by later *git p4 sync* operations to know which p4 changes are new.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.96. git-pack-objects(1)

2024-07-02
Revision History
2024-07-02

NAME

git-pack-objects - Create a packed archive of objects

SYNOPSIS

git pack-objects [-q | --progress | --all-progress] [--all-progress-implied]
[--no-reuse-delta] [--delta-base-offset] [--non-empty]
[--local] [--incremental] [--window=<n>] [--depth=<n>]
[--revs [--unpacked | --all]] [--keep-pack=<pack-name>]
[--cruft] [--cruft-expiration=<time>]
[--stdout [--filter=<filter-spec>] | <base-name>]
[--shallow] [--keep-true-parents] [--[no-]sparse] < <object-list>

DESCRIPTION

Reads list of objects from the standard input, and writes either one or more packed archives with the specified base-name to disk, or a packed archive to the standard output.

A packed archive is an efficient way to transfer a set of objects between two repositories as well as an access efficient archival format. In a packed archive, an object is either stored as a compressed whole or as a difference from some other object. The latter is often called a delta.

The packed archive format (.pack) is designed to be self-contained so that it can be unpacked without any further information. Therefore, each object that a delta depends upon must be present within the pack.

A pack index file (.idx) is generated for fast, random access to the objects in the pack. Placing both the index file (.idx) and the packed archive (.pack) in the pack/ subdirectory of $GIT_OBJECT_DIRECTORY (or any of the directories on $GIT_ALTERNATE_OBJECT_DIRECTORIES) enables Git to read from the pack archive.

The *git unpack-objects* command can read the packed archive and expand the objects contained in the pack into "one-file one-object" format; this is typically done by the smart-pull commands when a pack is created on-the-fly for efficient network transport by their peers.

OPTIONS

base-name

Write into pairs of files (.pack and .idx), using <base-name> to determine the name of the created file. When this option is used, the two files in a pair are written in <base-name>-<SHA-1>.{pack,idx} files. <SHA-1> is a hash based on the pack content and is written to the standard output of the command.

--stdout

Write the pack contents (what would have been written to .pack file) out to the standard output.

--revs

Read the revision arguments from the standard input, instead of individual object names. The revision arguments are processed the same way as *git rev-list* with the *--objects* flag uses its *commit* arguments to build the list of objects it outputs. The objects on the resulting list are packed. Besides revisions, *--not* or *--shallow <SHA-1>* lines are also accepted.

--unpacked

This implies *--revs*. When processing the list of revision arguments read from the standard input, limit the objects packed to those that are not already packed.

--all

This implies *--revs*. In addition to the list of revision arguments read from the standard input, pretend as if all refs under *refs/* are specified to be included.

--include-tag

Include unasked-for annotated tags if the object they reference was included in the resulting packfile. This can be useful to send new tags to native Git clients.

--stdin-packs

Read the basenames of packfiles (e.g., *pack-1234abcd.pack*) from the standard input, instead of object names or revision arguments. The resulting pack contains all objects listed in the included packs (those not beginning with ^), excluding any objects listed in the excluded packs (beginning with ^).

Incompatible with *--revs*, or options that imply *--revs* (such as *--all*), with the exception of *--unpacked*, which is compatible.

--cruft

Packs unreachable objects into a separate "cruft" pack, denoted by the existence of a *.mtimes* file. Typically used by *git repack --cruft*. Callers provide a list of pack names and indicate which packs will remain in the repository, along with which packs will be deleted (indicated by the - prefix). The contents of the cruft pack are all objects not contained in the surviving packs which have not exceeded the grace period (see *--cruft-expiration* below), or which have exceeded the grace period, but are reachable from an other object which hasn't.

When the input lists a pack containing all reachable objects (and lists all other packs as pending deletion), the corresponding cruft pack will contain all unreachable objects (with mtime newer than the *--cruft-expiration*) along with any unreachable objects whose mtime is older than the *--cruft-expiration*, but are reachable from an unreachable object whose mtime is newer than the *--cruft-expiration*).

Incompatible with *--unpack-unreachable*, *--keep-unreachable*, *--pack-loose-unreachable*, *--stdin-packs*, as well as any other options which imply *--revs*.

--cruft-expiration=<approxidate>

If specified, objects are eliminated from the cruft pack if they have an mtime older than *<approxidate>*. If unspecified (and given *--cruft*), then no objects are eliminated.

--window=<n> , --depth=<n>

These two options affect how the objects contained in the pack are stored using delta compression. The objects are first internally sorted by type, size and optionally names and compared against the other objects within --window to see if using delta compression saves space. --depth limits the maximum delta depth; making it too deep affects the performance on the unpacker side, because delta data needs to be applied that many times to get to the necessary object.

The default value for --window is 10 and --depth is 50. The maximum depth is 4095.

--window-memory=<n>

This option provides an additional limit on top of *--window*; the window size will dynamically scale down so as to not take up more than *<n>* bytes in memory. This is useful in repositories with a mix of large and small objects to not run out of memory with a large window, but still be able to take advantage of the large window for the smaller objects. The size can be suffixed with "k", "m", or "g". *--window-memory=0* makes memory usage unlimited. The default is taken from the *pack.windowMemory* configuration variable.

--max-pack-size=<n>

In unusual scenarios, you may not be able to create files larger than a certain size on your filesystem, and this option can be used to tell the command to split the output packfile into multiple independent packfiles, each not larger than the given size. The size can be suffixed with "k", "m", or "g". The minimum size allowed is limited to 1 MiB. The default is unlimited, unless the config variable *pack.packSizeLimit* is set. Note that this option may result in a larger and slower repository; see the discussion in *pack.packSizeLimit*.

--honor-pack-keep

This flag causes an object already in a local pack that has a .keep file to be ignored, even if it would have otherwise been packed.

--keep-pack=<pack-name>

This flag causes an object already in the given pack to be ignored, even if it would have otherwise been packed. *<pack-name>* is the pack file name without leading directory (e.g. *pack-123.pack*). The option could be specified multiple times to keep multiple packs.

--incremental

This flag causes an object already in a pack to be ignored even if it would have otherwise been packed.

--local

This flag causes an object that is borrowed from an alternate object store to be ignored even if it would have otherwise been packed.

--non-empty

Only create a packed archive if it would contain at least one object.

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

--all-progress

When --stdout is specified then progress report is displayed during the object count and compression phases but inhibited during the write-out phase. The reason is that in some cases the output stream is directly linked to another command which may wish to display progress status of its own as it processes incoming pack data. This flag is like --progress except that it forces progress report for the write-out phase as well even if --stdout is used.

--all-progress-implied

This is used to imply --all-progress whenever progress display is activated. Unlike --all-progress this flag doesn't actually force any progress display by itself.

-q

This flag makes the command not to report its progress on the standard error stream.

--no-reuse-delta

When creating a packed archive in a repository that has existing packs, the command reuses existing deltas. This sometimes results in a slightly suboptimal pack. This flag tells the command not to reuse existing deltas but compute them from scratch.

--no-reuse-object

This flag tells the command not to reuse existing object data at all, including non deltified object, forcing recompression of everything. This implies --no-reuse-delta. Useful only in the obscure case where wholesale enforcement of a different compression level on the packed data is desired.

--compression=<n>

Specifies compression level for newly-compressed data in the generated pack. If not specified, pack compression level is determined first by pack.compression, then by core.compression, and defaults to -1, the zlib default, if neither is set. Add --no-reuse-object if you want to force a uniform compression level on all data no matter the source.

--[no-]sparse

Toggle the "sparse" algorithm to determine which objects to include in the pack, when combined with the "--revs" option. This algorithm only walks trees that appear in paths that introduce new objects. This can have significant performance benefits when computing a pack to send a small change. However, it is possible that extra objects are added to the pack-file if the included commits contain certain types of direct renames. If this option is not included, it defaults to the value of *pack.useSparse*, which is true unless otherwise specified.

--thin

Create a "thin" pack by omitting the common objects between a sender and a receiver in order to reduce network transfer. This option only makes sense in conjunction with --stdout.

Note: A thin pack violates the packed archive format by omitting required objects and is thus unusable by Git without making it self-contained. Use *git index-pack --fix-thin* (see Section G.3.69, “git-index-pack(1)”) to restore the self-contained property.

--shallow

Optimize a pack that will be provided to a client with a shallow repository. This option, combined with --thin, can result in a smaller pack at the cost of speed.

--delta-base-offset

A packed archive can express the base object of a delta as either a 20-byte object name or as an offset in the stream, but ancient versions of Git don't understand the latter. By default, *git pack-objects* only uses the former format for better compatibility. This option allows the command to use the latter format for compactness.

Depending on the average delta chain length, this option typically shrinks the resulting packfile by 3-5 percent.

Note: Porcelain commands such as *git gc* (see Section G.3.58, “git-gc(1)”), *git repack* (see Section G.3.113, “git-repack(1)”) pass this option by default in modern Git when they put objects in your repository into pack files. So does *git bundle* (see Section G.3.12, “git-bundle(1)”) when it creates a bundle.

--threads=<n>

Specifies the number of threads to spawn when searching for best delta matches. This requires that pack-objects be compiled with pthreads otherwise this option is ignored with a warning. This is meant to reduce packing time on multiprocessor machines. The required amount of memory for the delta search window is however multiplied by the number of threads. Specifying 0 will cause Git to auto-detect the number of CPU's and set the number of threads accordingly.

--index-version=<version>[,<offset>]

This is intended to be used by the test suite only. It allows to force the version for the generated pack index, and to force 64-bit index entries on objects located above the given offset.

--keep-true-parents

With this option, parents that are hidden by grafts are packed nevertheless.

--filter=<filter-spec>

Omits certain objects (usually blobs) from the resulting packfile. See Section G.3.120, “git-rev-list(1)” for valid <*filter-spec*> forms.

--no-filter

Turns off any previous --*filter*= argument.

--missing=<missing-action>

A debug option to help with future "partial clone" development. This option specifies how missing objects are handled.

The form --*missing=error* requests that pack-objects stop with an error if a missing object is encountered. If the repository is a partial clone, an attempt to fetch missing objects will be made before declaring them missing. This is the default action.

The form --*missing=allow-any* will allow object traversal to continue if a missing object is encountered. No fetch of a missing object will occur. Missing objects will silently be omitted from the results.

The form --*missing=allow-promisor* is like *allow-any*, but will only allow object traversal to continue for EXPECTED promisor missing objects. No fetch of a missing object will occur. An unexpected missing object will raise an error.

--exclude-promisor-objects

Omit objects that are known to be in the promisor remote. (This option has the purpose of operating only on locally created objects, so that when we repack, we still maintain a distinction between locally created objects [without .promisor] and objects from the promisor remote [with .promisor].) This is used with partial clone.

--keep-unreachable

Objects unreachable from the refs in packs named with --unpacked= option are added to the resulting pack, in addition to the reachable objects that are not in packs marked with *.keep files. This implies --*revs*.

--pack-loose-unreachable

Pack unreachable loose objects (and their loose counterparts removed). This implies --*revs*.

--unpack-unreachable

Keep unreachable objects in loose form. This implies *--revs*.

--delta-islands

Restrict delta matches based on "islands". See DELTA ISLANDS below.

DELTA ISLANDS

When possible, *pack-objects* tries to reuse existing on-disk deltas to avoid having to search for new ones on the fly. This is an important optimization for serving fetches, because it means the server can avoid inflating most objects at all and just send the bytes directly from disk. This optimization can't work when an object is stored as a delta against a base which the receiver does not have (and which we are not already sending). In that case the server "breaks" the delta and has to find a new one, which has a high CPU cost. Therefore it's important for performance that the set of objects in on-disk delta relationships match what a client would fetch.

In a normal repository, this tends to work automatically. The objects are mostly reachable from the branches and tags, and that's what clients fetch. Any deltas we find on the server are likely to be between objects the client has or will have.

But in some repository setups, you may have several related but separate groups of ref tips, with clients tending to fetch those groups independently. For example, imagine that you are hosting several "forks" of a repository in a single shared object store, and letting clients view them as separate repositories through *GIT_NAMESPACE* or separate repos using the alternates mechanism. A naive repack may find that the optimal delta for an object is against a base that is only found in another fork. But when a client fetches, they will not have the base object, and we'll have to find a new delta on the fly.

A similar situation may exist if you have many refs outside of *refs/heads/* and *refs/tags/* that point to related objects (e.g., *refs/pull* or *refs/changes* used by some hosting providers). By default, clients fetch only heads and tags, and deltas against objects found only in those other groups cannot be sent as-is.

Delta islands solve this problem by allowing you to group your refs into distinct "islands". Pack-objects computes which objects are reachable from which islands, and refuses to make a delta from an object *A* against a base which is not present in all of *A*'s islands. This results in slightly larger packs (because we miss some delta opportunities), but guarantees that a fetch of one island will not have to recompute deltas on the fly due to crossing island boundaries.

When repacking with delta islands the delta window tends to get clogged with candidates that are forbidden by the config. Repacking with a big --window helps (and doesn't take as long as it otherwise might because we can reject some object pairs based on islands before doing any computation on the content).

Islands are configured via the *pack.island* option, which can be specified multiple times. Each value is a left-anchored regular expressions matching refnames. For example:

```
[pack]
island = refs/heads/
island = refs/tags/
```

puts heads and tags into an island (whose name is the empty string; see below for more on naming). Any refs which do not match those regular expressions (e.g., *refs/pull/123*) is not in any island. Any object which is reachable only from *refs/pull/* (but not heads or tags) is therefore not a candidate to be used as a base for *refs/heads/*.

Refs are grouped into islands based on their "names", and two regexes that produce the same name are considered to be in the same island. The names are computed from the regexes by concatenating any capture groups from the regex, with a - dash in between. (And if there are no capture groups, then the name is the empty string, as in the above example.) This allows you to create arbitrary numbers of islands. Only up to 14 such capture groups are supported though.

For example, imagine you store the refs for each fork in *refs/virtual/ID*, where *ID* is a numeric identifier. You might then configure:

```
[pack]
island = refs/virtual/([0-9]+)/heads/
island = refs/virtual/([0-9]+)/tags/
island = refs/virtual/([0-9]+)/(pull)/
```

That puts the heads and tags for each fork in their own island (named "1234" or similar), and the pull refs for each go into their own "1234-pull".

Note that we pick a single island for each regex to go into, using "last one wins" ordering (which allows repo-specific config to take precedence over user-wide config, and so forth).

CONFIGURATION

Various configuration variables affect packing, see Section G.3.29, “git-config(1)” (search for "pack" and "delta").

Notably, delta compression is not used on objects larger than the *core.bigFileThreshold* configuration variable and on files with the attribute *delta* set to false.

SEE ALSO

Section G.3.120, “git-rev-list(1)” Section G.3.113, “git-repack(1)” Section G.3.100, “git-prune-packed(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.97. git-pack-redundant(1)

2024-07-02

Revision History

2024-07-02

NAME

git-pack-redundant - Find redundant pack files

SYNOPSIS

git pack-redundant [--verbose] [--alt-odb] (--all | <pack-filename>…)

WARNING

git pack-redundant has been deprecated and is scheduled for removal in a future version of Git. Because it can only remove entire duplicate packs and not individual duplicate objects, it is generally not a useful tool for reducing repository size. You are better off using *git gc* to do so, which will put objects into a new pack, removing duplicates.

Running *pack-redundant* without the *--i-still-use-this* flag will fail in this release. If you believe you have a use case for which *pack-redundant* is better suited and oppose this removal, please contact the Git mailing list at git@vger.kernel.org [mailto:git@vger.kernel.org]. More information about the list is available at https://git-scm.com/community.

DESCRIPTION

This program computes which packs in your repository are redundant. The output is suitable for piping to *xargs rm* if you are in the root of the repository.

git pack-redundant accepts a list of objects on standard input. Any objects given will be ignored when checking which packs are required. This makes the following command useful when wanting to remove packs which contain unreachable objects.

git fsck --full --unreachable | cut -d ' ' -f3 | \ git pack-redundant --all | xargs rm

OPTIONS

--all

Processes all packs. Any filenames on the command line are ignored.

--alt-odb

Don't require objects present in packs from alternate object database (odb) directories to be present in local packs.

--verbose

Outputs some statistics to stderr. Has a small performance penalty.

SEE ALSO

Section G.3.96, “git-pack-objects(1)” Section G.3.113, “git-repack(1)” Section G.3.100, “git-prune-packed(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.98. git-pack-refs(1)

2024-07-02
Revision History
2024-07-02

NAME

git-pack-refs - Pack heads and tags for efficient repository access

SYNOPSIS

git pack-refs [--all] [--no-prune] [--auto] [--include <pattern>] [--exclude <pattern>]

DESCRIPTION

Traditionally, tips of branches and tags (collectively known as *refs*) were stored one file per ref in a (sub)directory under *$GIT_DIR/refs* directory. While many branch tips tend to be updated often, most tags and some branch tips are never updated. When a repository has hundreds or thousands of tags, this one-file-per-ref format both wastes storage and hurts performance.

This command is used to solve the storage and performance problem by storing the refs in a single file, *$GIT_DIR/packed-refs*. When a ref is missing from the traditional *$GIT_DIR/refs* directory hierarchy, it is looked up in this file and used if found.

Subsequent updates to branches always create new files under *$GIT_DIR/refs* directory hierarchy.

A recommended practice to deal with a repository with too many refs is to pack its refs with *--all* once, and occasionally run *git pack-refs*. Tags are by definition stationary and are not expected to change. Branch heads will be packed with the initial *pack-refs --all*, but only the currently active branch heads will become unpacked, and the next *pack-refs* (without *--all*) will leave them unpacked.

OPTIONS

--all

The command by default packs all tags and refs that are already packed, and leaves other refs alone. This is because branches are expected to be actively developed and packing their tips does not help performance. This option causes all refs to be packed as well, with the exception of hidden refs, broken refs, and symbolic refs. Useful for a repository with many branches of historical interests.

--no-prune

The command usually removes loose refs under *$GIT_DIR/refs* hierarchy after packing them. This option tells it not to.

--auto

Pack refs as needed depending on the current state of the ref database. The behavior depends on the ref format used by the repository and may change in the future.

- "files": No special handling for *--auto* has been implemented.
- "reftable": Tables are compacted such that they form a geometric sequence. For two tables N and N+1, where N+1 is newer, this maintains the property that N is at least twice as big as N+1. Only tables that violate this property are compacted.

--include <pattern>

Pack refs based on a *glob(7)* pattern. Repetitions of this option accumulate inclusion patterns. If a ref is both included in *--include* and *--exclude*, *--exclude* takes precedence. Using *--include* will preclude all tags from being included by default. Symbolic refs and broken refs will never be packed. When used with *--all*, it will be a noop. Use *--no-include* to clear and reset the list of patterns.

--exclude <pattern>

Do not pack refs matching the given *glob(7)* pattern. Repetitions of this option accumulate exclusion patterns. Use *--no-exclude* to clear and reset the list of patterns. If a ref is already packed, including it with *--exclude* will not unpack it.

When used with *--all*, pack only loose refs which do not match any of the provided *--exclude* patterns.

When used with *--include*, refs provided to *--include*, minus refs that are provided to *--exclude* will be packed.

BUGS

Older documentation written before the packed-refs mechanism was introduced may still say things like ".git/refs/heads/<branch> file exists" when it means "branch <branch> exists".

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.99. git-patch-id(1)

2024-07-02
Revision History
2024-07-02

NAME

git-patch-id - Compute unique ID for a patch

SYNOPSIS

git patch-id [--stable | --unstable | --verbatim]

DESCRIPTION

Read a patch from the standard input and compute the patch ID for it.

A "patch ID" is nothing but a sum of SHA-1 of the file diffs associated with a patch, with line numbers ignored. As such, it's "reasonably stable", but at the same time also reasonably unique, i.e., two patches that have the same "patch ID" are almost guaranteed to be the same thing.

The main usecase for this command is to look for likely duplicate commits.

When dealing with *git diff-tree* output, it takes advantage of the fact that the patch is prefixed with the object name of the commit, and outputs two 40-byte hexadecimal strings. The first string is the patch ID, and the second string is the commit ID. This can be used to make a mapping from patch ID to commit ID.

OPTIONS

--verbatim

Calculate the patch-id of the input as it is given, do not strip any whitespace.

`This is the default if patchid.verbatim is true.`

--stable

Use a "stable" sum of hashes as the patch ID. With this option:

- Reordering file diffs that make up a patch does not affect the ID. In particular, two patches produced by comparing the same two trees with two different settings for "-O<orderfile>" result in the same patch ID signature, thereby allowing the computed result to be used as a key to index some meta-information about the change between the two trees;
- Result is different from the value produced by git 1.9 and older or produced when an "unstable" hash (see --unstable below) is configured - even when used on a diff output taken without any use of "-O<orderfile>", thereby making existing databases storing such "unstable" or historical patch-ids unusable.
- All whitespace within the patch is ignored and does not affect the id.

  `This is the default if patchid.stable is set to true.`

--unstable

Use an "unstable" hash as the patch ID. With this option, the result produced is compatible with the patch-id value produced by git 1.9 and older and whitespace is ignored. Users with pre-existing databases storing patch-ids produced by git 1.9 and older (who do not deal with reordered patches) may want to use this option.

`This is the default.`

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.100. git-prune-packed(1)

2024-07-02

Revision History

2024-07-02

NAME

git-prune-packed - Remove extra objects that are already in pack files

SYNOPSIS

git prune-packed [-n | --dry-run] [-q | --quiet]

DESCRIPTION

This program searches the *$GIT_OBJECT_DIRECTORY* for all objects that currently exist in a pack file as well as in the independent object directories.

All such extra objects are removed.

A pack is a collection of objects, individually compressed, with delta compression applied, stored in a single file, with an associated index file.

Packs are used to reduce the load on mirror systems, backup engines, disk storage, etc.

OPTIONS

-n , --dry-run

Don't actually remove any objects, only show those that would have been removed.

-q , --quiet

Squelch the progress indicator.

SEE ALSO

Section G.3.96, “git-pack-objects(1)” Section G.3.113, “git-repack(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.101. git-prune(1)

2024-07-02
Revision History
2024-07-02

NAME

git-prune - Prune all unreachable objects from the object database

SYNOPSIS

git prune [-n] [-v] [--progress] [--expire <time>] [--] [<head>…]

DESCRIPTION

> **Note**
>
> In most cases, users should run *git gc*, which calls *git prune*. See the section "NOTES", below.

This runs *git fsck --unreachable* using all the refs available in *refs/*, optionally with an additional set of objects specified on the command line, and prunes all unpacked objects unreachable from any of these head objects from the object database. In addition, it prunes the unpacked objects that are also found in packs by running *git prune-packed*. It also removes entries from .git/shallow that are not reachable by any ref.

Note that unreachable, packed objects will remain. If this is not desired, see Section G.3.113, “git-repack(1)”.

OPTIONS

-n , --dry-run

Do not remove anything; just report what it would remove.

-v , --verbose

Report all removed objects.

--progress

Show progress.

--expire <time>

Only expire loose objects older than <time>.

--

Do not interpret any more arguments as options.

<head>…

In addition to objects reachable from any of our references, keep objects reachable from listed <head>s.

EXAMPLES

To prune objects not used by your repository or another that borrows from your repository via its *.git/objects/info/alternates*:

```
$ git prune $(cd ../another && git rev-parse --all)
```

NOTES

In most cases, users will not need to call *git prune* directly, but should instead call *git gc*, which handles pruning along with many other housekeeping tasks.

For a description of which objects are considered for pruning, see *git fsck*'s --unreachable option.

SEE ALSO

Section G.3.56, “git-fsck(1)”, Section G.3.58, “git-gc(1)”, Section G.3.109, “git-reflog(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.102. git-pull(1)

2024-07-02

Revision History

2024-07-02

NAME

git-pull - Fetch from and integrate with another repository or a local branch

SYNOPSIS

git pull [<options>] [<repository> [<refspec>…]]

DESCRIPTION

Incorporates changes from a remote repository into the current branch. If the current branch is behind the remote, then by default it will fast-forward the current branch to match the remote. If the current branch and the remote have diverged, the user needs to specify how to reconcile the divergent branches with *--rebase* or *--no-rebase* (or the corresponding configuration option in *pull.rebase*).

More precisely, *git pull* runs *git fetch* with the given parameters and then depending on configuration options or command line flags, will call either *git rebase* or *git merge* to reconcile diverging branches.

<repository> should be the name of a remote repository as passed to Section G.3.49, “git-fetch(1)”. <refspec> can name an arbitrary remote ref (for example, the name of a tag) or even a collection of refs with corresponding remote-tracking branches (e.g., refs/heads/*:refs/remotes/origin/*), but usually it is the name of a branch in the remote repository.

Default values for <repository> and <branch> are read from the "remote" and "merge" configuration for the current branch as set by Section G.3.10, “git-branch(1)” *--track*.

Assume the following history exists and the current branch is "*master*":

```
          A---B---C master on origin
         /
    D---E---F---G master
        ^
        origin/master in your repository
```

Then "*git pull*" will fetch and replay the changes from the remote *master* branch since it diverged from the local *master* (i.e., *E*) until its current commit (*C*) on top of *master* and record the result in a new commit along with the names of the two parent commits and a log message from the user describing the changes.

```
          A---B---C origin/master
         /         \
    D---E---F---G---H master
```

See Section G.3.86, “git-merge(1)” for details, including how conflicts are presented and handled.

In Git 1.7.0 or later, to cancel a conflicting merge, use *git reset --merge*. **Warning**: In older versions of Git, running *git pull* with uncommitted changes is discouraged: while possible, it leaves you in a state that may be hard to back out of in the case of a conflict.

If any of the remote changes overlap with local uncommitted changes, the merge will be automatically canceled and the work tree untouched. It is generally best to get any local changes in working order before pulling or stash them away with Section G.3.137, “git-stash(1)”.

OPTIONS

-q , --quiet

This is passed to both underlying git-fetch to squelch reporting of during transfer, and underlying git-merge to squelch output during merging.

-v , --verbose

Pass --verbose to git-fetch and git-merge.

--[no-]recurse-submodules[=(yes|on-demand|no)]

This option controls if new commits of populated submodules should be fetched, and if the working trees of active submodules should be updated, too (see Section G.3.49, “git-fetch(1)”, Section G.3.29, “git-config(1)” and Section G.4.10, “gitmodules(5)”).

If the checkout is done via rebase, local submodule commits are rebased as well.

If the update is done via merge, the submodule conflicts are resolved and checked out.

1. Options related to merging

--commit , --no-commit

Perform the merge and commit the result. This option can be used to override --no-commit. Only useful when merging.

With --no-commit perform the merge and stop just before creating a merge commit, to give the user a chance to inspect and further tweak the merge result before committing.

Note that fast-forward updates do not create a merge commit and therefore there is no way to stop those merges with --no-commit. Thus, if you want to ensure your branch is not changed or updated by the merge command, use --no-ff with --no-commit.

--edit , -e , --no-edit

Invoke an editor before committing successful mechanical merge to further edit the auto-generated merge message, so that the user can explain and justify the merge. The *--no-edit* option can be used to accept the auto-generated message (this is generally discouraged).

Older scripts may depend on the historical behaviour of not allowing the user to edit the merge log message. They will see an editor opened when they run *git merge*. To make it easier to adjust such scripts to the updated behaviour, the environment variable *GIT_MERGE_AUTOEDIT* can be set to *no* at the beginning of them.

--cleanup=<mode>

This option determines how the merge message will be cleaned up before committing. See Section G.3.28, “git-commit(1)” for more details. In addition, if the *<mode>* is given a value of *scissors*, scissors will be appended to *MERGE_MSG* before being passed on to the commit machinery in the case of a merge conflict.

--ff-only

Only update to the new history if there is no divergent local history. This is the default when no method for reconciling divergent histories is provided (via the --rebase=* flags).

--ff , --no-ff

When merging rather than rebasing, specifies how a merge is handled when the merged-in history is already a descendant of the current history. If merging is requested, *--ff* is the default unless merging an annotated (and possibly signed) tag that is not stored in its natural place in the *refs/tags/* hierarchy, in which case *--no-ff* is assumed.

With *--ff*, when possible resolve the merge as a fast-forward (only update the branch pointer to match the merged branch; do not create a merge commit). When not possible (when the merged-in history is not a descendant of the current history), create a merge commit.

With *--no-ff*, create a merge commit in all cases, even when the merge could instead be resolved as a fast-forward.

-S[<keyid>] , --gpg-sign[=<keyid>] , --no-gpg-sign

GPG-sign the resulting merge commit. The *keyid* argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space. *--no-gpg-sign* is useful to countermand both *commit.gpgSign* configuration variable, and earlier *--gpg-sign*.

--log[=<n>] , --no-log

In addition to branch names, populate the log message with one-line descriptions from at most <n> actual commits that are being merged. See also Section G.3.51, “git-fmt-merge-msg(1)”. Only useful when merging.

With --no-log do not list one-line descriptions from the actual commits being merged.

--signoff , --no-signoff

Add a *Signed-off-by* trailer by the committer at the end of the commit log message. The meaning of a signoff depends on the project to which you're committing. For example, it may certify that the committer has the rights to submit the work under the project's license or agrees to some contributor representation, such as a Developer Certificate of Origin. (See https://developercertificate.org for the one used by the Linux kernel

and Git projects.) Consult the documentation or leadership of the project to which you're contributing to understand how the signoffs are used in that project.

The --no-signoff option can be used to countermand an earlier --signoff option on the command line.

--stat , -n , --no-stat

Show a diffstat at the end of the merge. The diffstat is also controlled by the configuration option merge.stat.

With -n or --no-stat do not show a diffstat at the end of the merge.

--squash , --no-squash

Produce the working tree and index state as if a real merge happened (except for the merge information), but do not actually make a commit, move the *HEAD*, or record *$GIT_DIR/MERGE_HEAD* (to cause the next *git commit* command to create a merge commit). This allows you to create a single commit on top of the current branch whose effect is the same as merging another branch (or more in case of an octopus).

With --no-squash perform the merge and commit the result. This option can be used to override --squash.

With --squash, --commit is not allowed, and will fail.

Only useful when merging.

--[no-]verify

By default, the pre-merge and commit-msg hooks are run. When *--no-verify* is given, these are bypassed. See also Section G.4.7, “githooks(5)”. Only useful when merging.

-s <strategy> , --strategy=<strategy>

Use the given merge strategy; can be supplied more than once to specify them in the order they should be tried. If there is no *-s* option, a built-in list of strategies is used instead (*ort* when merging a single head, *octopus* otherwise).

-X <option> , --strategy-option=<option>

Pass merge strategy specific option through to the merge strategy.

--verify-signatures , --no-verify-signatures

Verify that the tip commit of the side branch being merged is signed with a valid key, i.e. a key that has a valid uid: in the default trust model, this means the signing key has been signed by a trusted key. If the tip commit of the side branch is not signed with a valid key, the merge is aborted.

Only useful when merging.

--summary , --no-summary

Synonyms to --stat and --no-stat; these are deprecated and will be removed in the future.

--autostash , --no-autostash

Automatically create a temporary stash entry before the operation begins, record it in the ref *MERGE_AUTOSTASH* and apply it after the operation ends. This means that you can run the operation on a dirty worktree. However, use with care: the final stash application after a successful merge might result in non-trivial conflicts.

--allow-unrelated-histories

By default, *git merge* command refuses to merge histories that do not share a common ancestor. This option can be used to override this safety when merging histories of two projects that started their lives independently. As that is a very rare occasion, no configuration variable to enable this by default exists and will not be added.

Only useful when merging.

-r , --rebase[=(false|true|merges|interactive)]

When true, rebase the current branch on top of the upstream branch after fetching. If there is a remote-tracking branch corresponding to the upstream branch and the upstream branch was rebased since last fetched, the rebase uses that information to avoid rebasing non-local changes.

When set to *merges*, rebase using *git rebase --rebase-merges* so that the local merge commits are included in the rebase (see Section G.3.107, “git-rebase(1)” for details).

When false, merge the upstream branch into the current branch.

When *interactive*, enable the interactive mode of rebase.

See *pull.rebase*, *branch.<name>.rebase* and *branch.autoSetupRebase* in Section G.3.29, “git-config(1)” if you want to make *git pull* always use *--rebase* instead of merging.

> **Note**
>
> This is a potentially *dangerous* mode of operation. It rewrites history, which does not bode well when you published that history already. Do **not** use this option unless you have read Section G.3.107, “git-rebase(1)” carefully.

--no-rebase

This is shorthand for --rebase=false.

2. Options related to fetching

--[no-]all

Fetch all remotes. This overrides the configuration variable *fetch.all*.

-a , --append

Append ref names and object names of fetched refs to the existing contents of *.git/FETCH_HEAD*. Without this option old data in *.git/FETCH_HEAD* will be overwritten.

--atomic

Use an atomic transaction to update local refs. Either all refs are updated, or on error, no refs are updated.

--depth=<depth>

Limit fetching to the specified number of commits from the tip of each remote branch history. If fetching to a *shallow* repository created by *git clone* with *--depth=<depth>* option (see Section G.3.24, “git-clone(1)”), deepen or shorten the history to the specified number of commits. Tags for the deepened commits are not fetched.

--deepen=<depth>

Similar to --depth, except it specifies the number of commits from the current shallow boundary instead of from the tip of each remote branch history.

--shallow-since=<date>

Deepen or shorten the history of a shallow repository to include all reachable commits after <date>.

--shallow-exclude=<revision>

Deepen or shorten the history of a shallow repository to exclude commits reachable from a specified remote branch or tag. This option can be specified multiple times.

--unshallow

If the source repository is complete, convert a shallow repository to a complete one, removing all the limitations imposed by shallow repositories.

If the source repository is shallow, fetch as much as possible so that the current repository has the same history as the source repository.

--update-shallow

By default when fetching from a shallow repository, *git fetch* refuses refs that require updating .git/shallow. This option updates .git/shallow and accepts such refs.

--negotiation-tip=<commit|glob>

By default, Git will report, to the server, commits reachable from all local refs to find common commits in an attempt to reduce the size of the to-be-received packfile. If specified, Git will only report commits reachable from the given tips. This is useful to speed up fetches when the user knows which local ref is likely to have commits in common with the upstream ref being fetched.

This option may be specified more than once; if so, Git will report commits reachable from any of the given commits.

The argument to this option may be a glob on ref names, a ref, or the (possibly abbreviated) SHA-1 of a commit. Specifying a glob is equivalent to specifying this option multiple times, one for each matching ref name.

See also the *fetch.negotiationAlgorithm* and *push.negotiate* configuration variables documented in Section G.3.29, “git-config(1)”, and the *--negotiate-only* option below.

--negotiate-only

Do not fetch anything from the server, and instead print the ancestors of the provided *--negotiation-tip=** arguments, which we have in common with the server.

This is incompatible with *--recurse-submodules=[yes|on-demand]*. Internally this is used to implement the *push.negotiate* option, see Section G.3.29, “git-config(1)”.

--dry-run

Show what would be done, without making any changes.

--porcelain

Print the output to standard output in an easy-to-parse format for scripts. See section OUTPUT in Section G.3.49, “git-fetch(1)” for details.

This is incompatible with *--recurse-submodules=[yes|on-demand]* and takes precedence over the *fetch.output* config option.

-f , --force

When *git fetch* is used with *<src>:<dst>* refspec, it may refuse to update the local branch as discussed in the *<refspec>* part of the Section G.3.49, “git-fetch(1)” documentation. This option overrides that check.

-k , --keep

Keep downloaded pack.

--prefetch

Modify the configured refspec to place all refs into the *refs/prefetch/* namespace. See the *prefetch* task in Section G.3.80, “git-maintenance(1)”.

-p , --prune

Before fetching, remove any remote-tracking references that no longer exist on the remote. Tags are not subject to pruning if they are fetched only because of the default tag auto-following or due to a --tags option. However, if tags are fetched due to an explicit refspec (either on the command line or in the remote configuration, for example if the remote was cloned with the --mirror option), then they are also subject to pruning. Supplying *--prune-tags* is a shorthand for providing the tag refspec.

--no-tags

By default, tags that point at objects that are downloaded from the remote repository are fetched and stored locally. This option disables this automatic tag following. The default behavior for a remote may be specified with the remote.<name>.tagOpt setting. See Section G.3.29, “git-config(1)”.

--refmap=<refspec>

When fetching refs listed on the command line, use the specified refspec (can be given more than once) to map the refs to remote-tracking branches, instead of the values of *remote.*.fetch* configuration variables for the remote repository. Providing an empty *<refspec>* to the *--refmap* option causes Git to ignore the configured refspecs and rely entirely on the refspecs supplied as command-line arguments. See section on "Configured Remote-tracking Branches" for details.

-t , --tags

Fetch all tags from the remote (i.e., fetch remote tags *refs/tags/** into local tags with the same name), in addition to whatever else would otherwise be fetched. Using this option alone does not subject tags to pruning, even if --prune is used (though tags may be pruned anyway if they are also the destination of an explicit refspec; see *--prune*).

-j , --jobs=<n>

Number of parallel children to be used for all forms of fetching.

If the *--multiple* option was specified, the different remotes will be fetched in parallel. If multiple submodules are fetched, they will be fetched in parallel. To control them independently, use the config settings *fetch.parallel* and *submodule.fetchJobs* (see Section G.3.29, “git-config(1)”).

Typically, parallel recursive and multi-remote fetches will be faster. By default fetches are performed sequentially, not in parallel.

--set-upstream

If the remote is fetched successfully, add upstream (tracking) reference, used by argument-less Section G.3.102, “git-pull(1)” and other commands. For more information, see *branch.<name>.merge* and *branch.<name>.remote* in Section G.3.29, “git-config(1)”.

--upload-pack <upload-pack>

When given, and the repository to fetch from is handled by *git fetch-pack*, *--exec=<upload-pack>* is passed to the command to specify non-default path for the command run on the other end.

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

-o <option> , --server-option=<option>

Transmit the given string to the server when communicating using protocol version 2. The given string must not contain a NUL or LF character. The server's handling of server options, including unknown ones, is server-specific. When multiple *--server-option=<option>* are given, they are all sent to the other side in the order listed on the command line.

--show-forced-updates

By default, git checks if a branch is force-updated during fetch. This can be disabled through fetch.showForcedUpdates, but the --show-forced-updates option guarantees this check occurs. See Section G.3.29, “git-config(1)”.

--no-show-forced-updates

By default, git checks if a branch is force-updated during fetch. Pass --no-show-forced-updates or set fetch.showForcedUpdates to false to skip this check for performance reasons. If used during *git-pull* the --ff-only option will still check for forced updates before attempting a fast-forward update. See Section G.3.29, “git-config(1)”.

-4 , --ipv4

Use IPv4 addresses only, ignoring IPv6 addresses.

-6 , --ipv6

Use IPv6 addresses only, ignoring IPv4 addresses.

<repository>

The "remote" repository that is the source of a fetch or pull operation. This parameter can be either a URL (see the section GIT URLS below) or the name of a remote (see the section REMOTES below).

<refspec>

Specifies which refs to fetch and which local refs to update. When no <refspec>s appear on the command line, the refs to fetch are read from *remote.<repository>.fetch* variables instead (see the section "CONFIGURED REMOTE-TRACKING BRANCHES" in Section G.3.49, “git-fetch(1)”).

The format of a <refspec> parameter is an optional plus +, followed by the source <src>, followed by a colon :, followed by the destination ref <dst>. The colon can be omitted when <dst> is empty. <src> is typically a ref, but it can also be a fully spelled hex object name.

A <refspec> may contain a * in its <src> to indicate a simple pattern match. Such a refspec functions like a glob that matches any ref with the same prefix. A pattern <refspec> must have a * in both the <src> and <dst>. It will map refs to the destination by replacing the * with the contents matched from the source.

If a refspec is prefixed by ^, it will be interpreted as a negative refspec. Rather than specifying which refs to fetch or which local refs to update, such a refspec will instead specify refs to exclude. A ref will be considered to match if it matches at least one positive refspec, and does not match any negative refspec. Negative refspecs can be useful to restrict the scope of a pattern refspec so that it will not include specific refs. Negative refspecs can themselves be pattern refspecs. However, they may only contain a <src> and do not specify a <dst>. Fully spelled out hex object names are also not supported.

tag <tag> means the same as *refs/tags/<tag>:refs/tags/<tag>*; it requests fetching everything up to the given tag.

The remote ref that matches <src> is fetched, and if <dst> is not an empty string, an attempt is made to update the local ref that matches it.

Whether that update is allowed without *--force* depends on the ref namespace it's being fetched to, the type of object being fetched, and whether the update is considered to be a fast-forward. Generally, the same rules apply for fetching as when pushing, see the *<refspec>...* section of Section G.3.103, “git-push(1)” for what those are. Exceptions to those rules particular to *git fetch* are noted below.

Until Git version 2.20, and unlike when pushing with Section G.3.103, “git-push(1)”, any updates to *refs/tags/** would be accepted without + in the refspec (or *--force*). When fetching, we promiscuously considered

all tag updates from a remote to be forced fetches. Since Git version 2.20, fetching to update *refs/tags/** works the same way as when pushing. I.e. any updates will be rejected without + in the refspec (or *--force*).

Unlike when pushing with Section G.3.103, "git-push(1)", any updates outside of *refs/{tags,heads}/** will be accepted without + in the refspec (or *--force*), whether that's swapping e.g. a tree object for a blob, or a commit for another commit that doesn't have the previous commit as an ancestor etc.

Unlike when pushing with Section G.3.103, "git-push(1)", there is no configuration which'll amend these rules, and nothing like a *pre-fetch* hook analogous to the *pre-receive* hook.

As with pushing with Section G.3.103, "git-push(1)", all of the rules described above about what's not allowed as an update can be overridden by adding an optional leading + to a refspec (or using the *--force* command line option). The only exception to this is that no amount of forcing will make the *refs/heads/** namespace accept a non-commit object.

Note

When the remote branch you want to fetch is known to be rewound and rebased regularly, it is expected that its new tip will not be a descendant of its previous tip (as stored in your remote-tracking branch the last time you fetched). You would want to use the + sign to indicate non-fast-forward updates will be needed for such branches. There is no way to determine or declare that a branch will be made available in a repository with this behavior; the pulling user simply must know this is the expected usage pattern for a branch.

Note

There is a difference between listing multiple <refspec> directly on *git pull* command line and having multiple *remote.<repository>.fetch* entries in your configuration for a <repository> and running a *git pull* command without any explicit <refspec> parameters. <refspec>s listed explicitly on the command line are always merged into the current branch after fetching. In other words, if you list more than one remote ref, *git pull* will create an Octopus merge. On the other hand, if you do not list any explicit <refspec> parameter on the command line, *git pull* will fetch all the <refspec>s it finds in the *remote.<repository>.fetch* configuration and merge only the first <refspec> found into the current branch. This is because making an Octopus from remote refs is rarely done, while keeping track of multiple remote heads in one-go by fetching more than one is often useful.

GIT URLS

In general, URLs contain information about the transport protocol, the address of the remote server, and the path to the repository. Depending on the transport protocol, some of this information may be absent.

Git supports ssh, git, http, and https protocols (in addition, ftp and ftps can be used for fetching, but this is inefficient and deprecated; do not use them).

The native transport (i.e. git:// URL) does no authentication and should be used with caution on unsecured networks.

The following syntaxes may be used with them:

- `ssh://[<user>@]<host>[:<port>]/<path-to-git-repo>`
- `git://<host>[:<port>]/<path-to-git-repo>`
- `http[s]://<host>[:<port>]/<path-to-git-repo>`
- `ftp[s]://<host>[:<port>]/<path-to-git-repo>`

An alternative scp-like syntax may also be used with the ssh protocol:

- [<*user*>@]<*host*>`:/`<*path-to-git-repo*>

This syntax is only recognized if there are no slashes before the first colon. This helps differentiate a local path that contains a colon. For example the local path *foo:bar* could be specified as an absolute path or *./foo:bar* to avoid being misinterpreted as an ssh url.

The ssh and git protocols additionally support `~`<*username*> expansion:

- `ssh://`[<*user*>@]<*host*>[`:`<*port*>]`/~`<*user*>`/`<*path-to-git-repo*>
- `git://`<*host*>[`:`<*port*>]`/~`<*user*>`/`<*path-to-git-repo*>
- [<*user*>@]<*host*>`:~`<*user*>`/`<*path-to-git-repo*>

For local repositories, also supported by Git natively, the following syntaxes may be used:

- */path/to/repo.git/*
- `file:///path/to/repo.git/`

These two syntaxes are mostly equivalent, except when cloning, when the former implies *--local* option. See Section G.3.24, “git-clone(1)” for details.

git clone, *git fetch* and *git pull*, but not *git push*, will also accept a suitable bundle file. See Section G.3.12, “git-bundle(1)”.

When Git doesn't know how to handle a certain transport protocol, it attempts to use the *remote-*<*transport*> remote helper, if one exists. To explicitly request a remote helper, the following syntax may be used:

- <*transport*>::<*address*>

where <*address*> may be a path, a server and path, or an arbitrary URL-like string recognized by the specific remote helper being invoked. See Section G.4.12, “gitremote-helpers(7)” for details.

If there are a large number of similarly-named remote repositories and you want to use a different format for them (such that the URLs you use will be rewritten into URLs that work), you can create a configuration section of the form:

```
[url "<actual-url-base>"]
    insteadOf = <other-url-base>
```

For example, with this:

```
[url "git://git.host.xz/"]
        insteadOf = host.xz:/path/to/
        insteadOf = work:
```

a URL like "work:repo.git" or like "host.xz:/path/to/repo.git" will be rewritten in any context that takes a URL to be "git://git.host.xz/repo.git".

If you want to rewrite URLs for push only, you can create a configuration section of the form:

```
[url "<actual-url-base>"]
    pushInsteadOf = <other-url-base>
```

For example, with this:

```
[url "ssh://example.org/"]
        pushInsteadOf = git://example.org/
```

a URL like "git://example.org/path/to/repo.git" will be rewritten to "ssh://example.org/path/to/repo.git" for pushes, but pulls will still use the original URL.

REMOTES

The name of one of the following can be used instead of a URL as <*repository*> argument:

- a remote in the Git configuration file: *$GIT_DIR/config*,
- a file in the *$GIT_DIR/remotes* directory, or
- a file in the *$GIT_DIR/branches* directory.

All of these also allow you to omit the refspec from the command line because they each contain a refspec which git will use by default.

1. Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using Section G.3.112, “git-remote(1)”, Section G.3.29, “git-config(1)” or even by a manual edit to the *$GIT_DIR/config* file. The URL of this remote will be used to access the repository. The refspec of this remote will be used by default when you do not provide a refspec on the command line. The entry in the config file would appear like this:

```
[remote "<name>"]
        url = <URL>
        pushurl = <pushurl>
        push = <refspec>
        fetch = <refspec>
```

The <*pushurl*> is used for pushes only. It is optional and defaults to <*URL*>. Pushing to a remote affects all defined pushurls or all defined urls if no pushurls are defined. Fetch, however, will only fetch from the first defined url if multiple urls are defined.

2. Named file in *$GIT_DIR/remotes*

You can choose to provide the name of a file in *$GIT_DIR/remotes*. The URL in this file will be used to access the repository. The refspec in this file will be used as default when you do not provide a refspec on the command line. This file should have the following format:

```
URL: one of the above URL formats
Push: <refspec>
Pull: <refspec>
```

Push: lines are used by *git push* and *Pull:* lines are used by *git pull* and *git fetch*. Multiple *Push:* and *Pull:* lines may be specified for additional branch mappings.

3. Named file in *$GIT_DIR/branches*

You can choose to provide the name of a file in *$GIT_DIR/branches*. The URL in this file will be used to access the repository. This file should have the following format:

```
<URL>#<head>
```

<*URL*> is required; #<*head*> is optional.

Depending on the operation, git will use one of the following refspecs, if you don't provide one on the command line. <*branch*> is the name of this file in *$GIT_DIR/branches* and <*head*> defaults to *master*.

git fetch uses:

```
refs/heads/<head>:refs/heads/<branch>
```

git push uses:

```
HEAD:refs/heads/<head>
```

MERGE STRATEGIES

The merge mechanism (*git merge* and *git pull* commands) allows the backend *merge strategies* to be chosen with *-s* option. Some strategies can also take their own options, which can be passed by giving *-X<option>* arguments to *git merge* and/or *git pull*.

ort

This is the default merge strategy when pulling or merging one branch. This strategy can only resolve two heads using a 3-way merge algorithm. When there is more than one common ancestor that can be used for 3-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the 3-way merge. This has been reported to result in fewer merge conflicts without causing mismerges by tests done on actual merge commits taken from Linux 2.6 kernel development history. Additionally this strategy can detect and handle merges involving renames. It does not make use of detected copies. The name for this algorithm is an acronym ("Ostensibly Recursive's Twin") and came from the fact that it was written as a replacement for the previous default algorithm, *recursive*.

The *ort* strategy can take the following options:

ours

This option forces conflicting hunks to be auto-resolved cleanly by favoring *our* version. Changes from the other tree that do not conflict with our side are reflected in the merge result. For a binary file, the entire contents are taken from our side.

This should not be confused with the *ours* merge strategy, which does not even look at what the other tree contains at all. It discards everything the other tree did, declaring *our* history contains all that happened in it.

theirs

This is the opposite of *ours*; note that, unlike *ours*, there is no *theirs* merge strategy to confuse this merge option with.

ignore-space-change , ignore-all-space , ignore-space-at-eol , ignore-cr-at-eol

Treats lines with the indicated type of whitespace change as unchanged for the sake of a three-way merge. Whitespace changes mixed with other changes to a line are not ignored. See also Section G.3.44, “git-diff(1)” *-b*, *-w*, *--ignore-space-at-eol*, and *--ignore-cr-at-eol*.

- If *their* version only introduces whitespace changes to a line, *our* version is used;
- If *our* version introduces whitespace changes but *their* version includes a substantial change, *their* version is used;
- Otherwise, the merge proceeds in the usual way.

renormalize

This runs a virtual check-out and check-in of all three stages of a file when resolving a three-way merge. This option is meant to be used when merging branches with different clean filters or end-of-line normalization rules. See "Merging branches with differing checkin/checkout attributes" in Section G.4.2, “gitattributes(5)” for details.

no-renormalize

Disables the *renormalize* option. This overrides the *merge.renormalize* configuration variable.

find-renames[=<n>]

Turn on rename detection, optionally setting the similarity threshold. This is the default. This overrides the *merge.renames* configuration variable. See also Section G.3.44, “git-diff(1)” *--find-renames*.

rename-threshold=<n>

Deprecated synonym for *find-renames=<n>*.

subtree[=<path>]

This option is a more advanced form of *subtree* strategy, where the strategy makes a guess on how two trees must be shifted to match with each other when merging. Instead, the specified path is prefixed (or stripped from the beginning) to make the shape of two trees to match.

recursive

This can only resolve two heads using a 3-way merge algorithm. When there is more than one common ancestor that can be used for 3-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the 3-way merge. This has been reported to result in fewer merge conflicts without causing mismerges by tests done on actual merge commits taken from Linux 2.6 kernel development history. Additionally this can detect and handle merges involving renames. It does not make use of detected copies. This was the default strategy for resolving two heads from Git v0.99.9k until v2.33.0.

The *recursive* strategy takes the same options as *ort*. However, there are three additional options that *ort* ignores (not documented above) that are potentially useful with the *recursive* strategy:

patience

Deprecated synonym for *diff-algorithm=patience*.

diff-algorithm=[patience|minimal|histogram|myers]

Use a different diff algorithm while merging, which can help avoid mismerges that occur due to unimportant matching lines (such as braces from distinct functions). See also Section G.3.44, “git-diff(1)” *--diff-algorithm*. Note that *ort* specifically uses *diff-algorithm=histogram*, while *recursive* defaults to the *diff.algorithm* config setting.

no-renames

Turn off rename detection. This overrides the *merge.renames* configuration variable. See also Section G.3.44, “git-diff(1)” *--no-renames*.

resolve

This can only resolve two heads (i.e. the current branch and another branch you pulled from) using a 3-way merge algorithm. It tries to carefully detect criss-cross merge ambiguities. It does not handle renames.

octopus

This resolves cases with more than two heads, but refuses to do a complex merge that needs manual resolution. It is primarily meant to be used for bundling topic branch heads together. This is the default merge strategy when pulling or merging more than one branch.

ours

This resolves any number of heads, but the resulting tree of the merge is always that of the current branch head, effectively ignoring all changes from all other branches. It is meant to be used to supersede old development history of side branches. Note that this is different from the -Xours option to the *recursive* merge strategy.

subtree

This is a modified *ort* strategy. When merging trees A and B, if B corresponds to a subtree of A, B is first adjusted to match the tree structure of A, instead of reading the trees at the same level. This adjustment is also done to the common ancestor tree.

With the strategies that use 3-way merge (including the default, *ort*), if a change is made on both branches, but later reverted on one of the branches, that change will be present in the merged result; some people find this behavior

confusing. It occurs because only the heads and the merge base are considered when performing a merge, not the individual commits. The merge algorithm therefore considers the reverted change as no change at all, and substitutes the changed version instead.

DEFAULT BEHAVIOUR

Often people use *git pull* without giving any parameter. Traditionally, this has been equivalent to saying *git pull origin*. However, when configuration *branch.<name>.remote* is present while on branch *<name>*, that value is used instead of *origin*.

In order to determine what URL to use to fetch from, the value of the configuration *remote.<origin>.url* is consulted and if there is not any such variable, the value on the *URL:* line in *$GIT_DIR/remotes/<origin>* is used.

In order to determine what remote branches to fetch (and optionally store in the remote-tracking branches) when the command is run without any refspec parameters on the command line, values of the configuration variable *remote.<origin>.fetch* are consulted, and if there aren't any, *$GIT_DIR/remotes/<origin>* is consulted and its *Pull:* lines are used. In addition to the refspec formats described in the OPTIONS section, you can have a globbing refspec that looks like this:

```
refs/heads/*:refs/remotes/origin/*
```

A globbing refspec must have a non-empty RHS (i.e. must store what were fetched in remote-tracking branches), and its LHS and RHS must end with /*. The above specifies that all remote branches are tracked using remote-tracking branches in *refs/remotes/origin/* hierarchy under the same name.

The rule to determine which remote branch to merge after fetching is a bit involved, in order not to break backward compatibility.

If explicit refspecs were given on the command line of *git pull*, they are all merged.

When no refspec was given on the command line, then *git pull* uses the refspec from the configuration or *$GIT_DIR/remotes/<origin>*. In such cases, the following rules apply:

1. If *branch.<name>.merge* configuration for the current branch *<name>* exists, that is the name of the branch at the remote site that is merged.
2. If the refspec is a globbing one, nothing is merged.
3. Otherwise the remote branch of the first refspec is merged.

EXAMPLES

- Update the remote-tracking branches for the repository you cloned from, then merge one of them into your current branch:

  ```
  $ git pull
  $ git pull origin
  ```

  Normally the branch merged in is the HEAD of the remote repository, but the choice is determined by the branch.<name>.remote and branch.<name>.merge options; see Section G.3.29, “git-config(1)” for details.

- Merge into the current branch the remote branch *next*:

  ```
  $ git pull origin next
  ```

  This leaves a copy of *next* temporarily in FETCH_HEAD, and updates the remote-tracking branch *origin/next*. The same can be done by invoking fetch and merge:

  ```
  $ git fetch origin
  $ git merge origin/next
  ```

If you tried a pull which resulted in complex conflicts and would want to start over, you can recover with *git reset*.

SECURITY

The fetch and push protocols are not designed to prevent one side from stealing data from the other repository that was not intended to be shared. If you have private data that you need to protect from a malicious peer, your best option is to store it in another repository. This applies to both clients and servers. In particular, namespaces on a server are not effective for read access control; you should only grant read access to a namespace to clients that you would trust with read access to the entire repository.

The known attack vectors are as follows:

1. The victim sends "have" lines advertising the IDs of objects it has that are not explicitly intended to be shared but can be used to optimize the transfer if the peer also has them. The attacker chooses an object ID X to steal and sends a ref to X, but isn't required to send the content of X because the victim already has it. Now the victim believes that the attacker has X, and it sends the content of X back to the attacker later. (This attack is most straightforward for a client to perform on a server, by creating a ref to X in the namespace the client has access to and then fetching it. The most likely way for a server to perform it on a client is to "merge" X into a public branch and hope that the user does additional work on this branch and pushes it back to the server without noticing the merge.)

2. As in #1, the attacker chooses an object ID X to steal. The victim sends an object Y that the attacker already has, and the attacker falsely claims to have X and not Y, so the victim sends Y as a delta against X. The delta reveals regions of X that are similar to Y to the attacker.

BUGS

Using --recurse-submodules can only fetch new commits in already checked out submodules right now. When e.g. upstream added a new submodule in the just fetched commits of the superproject the submodule itself cannot be fetched, making it impossible to check out that submodule later without having to do a fetch again. This is expected to be fixed in a future Git version.

SEE ALSO

Section G.3.49, “git-fetch(1)”, Section G.3.86, “git-merge(1)”, Section G.3.29, “git-config(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.103. git-push(1)

2024-07-02
Revision History
2024-07-02

NAME

git-push - Update remote refs along with associated objects

SYNOPSIS

git push [--all | --branches | --mirror | --tags] [--follow-tags] [--atomic] [-n | --dry-run] [--receive-pack=<git-receive-pack>
[--repo=<repository>] [-f | --force] [-d | --delete] [--prune] [-q | --quiet] [-v | --verbose]
[-u | --set-upstream] [-o <string> | --push-option=<string>]
[--[no-]signed|--signed=(true|false|if-asked)]
[--force-with-lease[=<refname>[:<expect>]] [--force-if-includes]]
[--no-verify] [<repository> [<refspec>…]]

DESCRIPTION

Updates remote refs using local refs, while sending objects necessary to complete the given refs.

You can make interesting things happen to a repository every time you push into it, by setting up *hooks* there. See documentation for Section G.3.108, “git-receive-pack(1)”.

When the command line does not specify where to push with the *<repository>* argument, *branch.*.remote* configuration for the current branch is consulted to determine where to push. If the configuration is missing, it defaults to *origin*.

When the command line does not specify what to push with *<refspec>...* arguments or *--all*, *--mirror*, *--tags* options, the command finds the default *<refspec>* by consulting *remote.*.push* configuration, and if it is not found, honors *push.default* configuration to decide what to push (See Section G.3.29, "git-config(1)" for the meaning of *push.default*).

When neither the command-line nor the configuration specifies what to push, the default behavior is used, which corresponds to the *simple* value for *push.default*: the current branch is pushed to the corresponding upstream branch, but as a safety measure, the push is aborted if the upstream branch does not have the same name as the local one.

OPTIONS

<repository>

The "remote" repository that is the destination of a push operation. This parameter can be either a URL (see the section GIT URLS below) or the name of a remote (see the section REMOTES below).

<refspec>…

Specify what destination ref to update with what source object. The format of a <refspec> parameter is an optional plus +, followed by the source object <src>, followed by a colon *:*, followed by the destination ref <dst>.

The <src> is often the name of the branch you would want to push, but it can be any arbitrary "SHA-1 expression", such as *master~4* or *HEAD* (see Section G.4.14, "gitrevisions(7)").

The <dst> tells which ref on the remote side is updated with this push. Arbitrary expressions cannot be used here, an actual ref must be named. If *git push [<repository>]* without any *<refspec>* argument is set to update some ref at the destination with *<src>* with *remote.<repository>.push* configuration variable, *:<dst>* part can be omitted--such a push will update a ref that *<src>* normally updates without any *<refspec>* on the command line. Otherwise, missing *:<dst>* means to update the same ref as the *<src>*.

If <dst> doesn't start with *refs/* (e.g. *refs/heads/master*) we will try to infer where in *refs/** on the destination <repository> it belongs based on the type of <src> being pushed and whether <dst> is ambiguous.

- If <dst> unambiguously refers to a ref on the <repository> remote, then push to that ref.
- If <src> resolves to a ref starting with refs/heads/ or refs/tags/, then prepend that to <dst>.
- Other ambiguity resolutions might be added in the future, but for now any other cases will error out with an error indicating what we tried, and depending on the *advice.pushUnqualifiedRefname* configuration (see Section G.3.29, "git-config(1)") suggest what refs/ namespace you may have wanted to push to.

The object referenced by <src> is used to update the <dst> reference on the remote side. Whether this is allowed depends on where in *refs/** the <dst> reference lives as described in detail below, in those sections "update" means any modifications except deletes, which as noted after the next few sections are treated differently.

The *refs/heads/** namespace will only accept commit objects, and updates only if they can be fast-forwarded.

The *refs/tags/** namespace will accept any kind of object (as commits, trees and blobs can be tagged), and any updates to them will be rejected.

It's possible to push any type of object to any namespace outside of *refs/{tags,heads}/**. In the case of tags and commits, these will be treated as if they were the commits inside *refs/heads/** for the purposes of whether the update is allowed.

I.e. a fast-forward of commits and tags outside *refs/{tags,heads}/** is allowed, even in cases where what's being fast-forwarded is not a commit, but a tag object which happens to point to a new commit which is a fast-forward of the commit the last tag (or commit) it's replacing. Replacing a tag with an entirely different tag is also allowed, if it points to the same commit, as well as pushing a peeled tag, i.e. pushing the commit that existing tag object points to, or a new tag object which an existing commit points to.

Tree and blob objects outside of *refs/{tags,heads}/** will be treated the same way as if they were inside *refs/tags/**, any update of them will be rejected.

All of the rules described above about what's not allowed as an update can be overridden by adding an the optional leading + to a refspec (or using *--force* command line option). The only exception to this is that no amount of forcing will make the *refs/heads/** namespace accept a non-commit object. Hooks and configuration can also override or amend these rules, see e.g. *receive.denyNonFastForwards* in Section G.3.29, "git-config(1)" and *pre-receive* and *update* in Section G.4.7, "githooks(5)".

Pushing an empty <src> allows you to delete the <dst> ref from the remote repository. Deletions are always accepted without a leading + in the refspec (or *--force*), except when forbidden by configuration or hooks. See *receive.denyDeletes* in Section G.3.29, "git-config(1)" and *pre-receive* and *update* in Section G.4.7, "githooks(5)".

The special refspec *:* (or *+:* to allow non-fast-forward updates) directs Git to push "matching" branches: for every branch that exists on the local side, the remote side is updated if a branch of the same name already exists on the remote side.

tag <tag> means the same as *refs/tags/<tag>:refs/tags/<tag>*.

--all , --branches

Push all branches (i.e. refs under *refs/heads/*); cannot be used with other <refspec>.

--prune

Remove remote branches that don't have a local counterpart. For example a remote branch *tmp* will be removed if a local branch with the same name doesn't exist any more. This also respects refspecs, e.g. *git push --prune remote refs/heads/*:refs/tmp/** would make sure that remote *refs/tmp/foo* will be removed if *refs/heads/foo* doesn't exist.

--mirror

Instead of naming each ref to push, specifies that all refs under *refs/* (which includes but is not limited to *refs/heads/*, *refs/remotes/*, and *refs/tags/*) be mirrored to the remote repository. Newly created local refs will be pushed to the remote end, locally updated refs will be force updated on the remote end, and deleted refs will be removed from the remote end. This is the default if the configuration option *remote.<remote>.mirror* is set.

-n , --dry-run

Do everything except actually send the updates.

--porcelain

Produce machine-readable output. The output status line for each ref will be tab-separated and sent to stdout instead of stderr. The full symbolic names of the refs will be given.

-d , --delete

All listed refs are deleted from the remote repository. This is the same as prefixing all refs with a colon.

--tags

All refs under *refs/tags* are pushed, in addition to refspecs explicitly listed on the command line.

--follow-tags

Push all the refs that would be pushed without this option, and also push annotated tags in *refs/tags* that are missing from the remote but are pointing at commit-ish that are reachable from the refs being pushed. This can also be specified with configuration variable *push.followTags*. For more information, see *push.followTags* in Section G.3.29, “git-config(1)”.

--[no-]signed , --signed=(true|false|if-asked)

GPG-sign the push request to update refs on the receiving side, to allow it to be checked by the hooks and/or be logged. If *false* or *--no-signed*, no signing will be attempted. If *true* or *--signed*, the push will fail if the server does not support signed pushes. If set to *if-asked*, sign if and only if the server supports signed pushes. The push will also fail if the actual call to *gpg --sign* fails. See Section G.3.108, “git-receive-pack(1)” for the details on the receiving end.

--[no-]atomic

Use an atomic transaction on the remote side if available. Either all refs are updated, or on error, no refs are updated. If the server does not support atomic pushes the push will fail.

-o <option> , --push-option=<option>

Transmit the given string to the server, which passes them to the pre-receive as well as the post-receive hook. The given string must not contain a NUL or LF character. When multiple *--push-option=<option>* are given, they are all sent to the other side in the order listed on the command line. When no *--push-option=<option>* is given from the command line, the values of configuration variable *push.pushOption* are used instead.

--receive-pack=<git-receive-pack> , --exec=<git-receive-pack>

Path to the *git-receive-pack* program on the remote end. Sometimes useful when pushing to a remote repository over ssh, and you do not have the program in a directory on the default $PATH.

--[no-]force-with-lease , --force-with-lease=<refname> , --force-with-lease=<refname>:<expect>

Usually, "git push" refuses to update a remote ref that is not an ancestor of the local ref used to overwrite it.

This option overrides this restriction if the current value of the remote ref is the expected value. "git push" fails otherwise.

Imagine that you have to rebase what you have already published. You will have to bypass the "must fast-forward" rule in order to replace the history you originally published with the rebased history. If somebody else built on top of your original history while you are rebasing, the tip of the branch at the remote may advance with their commit, and blindly pushing with *--force* will lose their work.

This option allows you to say that you expect the history you are updating is what you rebased and want to replace. If the remote ref still points at the commit you specified, you can be sure that no other people did anything to the ref. It is like taking a "lease" on the ref without explicitly locking it, and the remote ref is updated only if the "lease" is still valid.

--force-with-lease alone, without specifying the details, will protect all remote refs that are going to be updated by requiring their current value to be the same as the remote-tracking branch we have for them.

--force-with-lease=<refname>, without specifying the expected value, will protect the named ref (alone), if it is going to be updated, by requiring its current value to be the same as the remote-tracking branch we have for it.

--force-with-lease=<refname>:<expect> will protect the named ref (alone), if it is going to be updated, by requiring its current value to be the same as the specified value *<expect>* (which is allowed to be different from the remote-tracking branch we have for the refname, or we do not even have to have such a remote-tracking branch when this form is used). If *<expect>* is the empty string, then the named ref must not already exist.

Note that all forms other than *--force-with-lease=<refname>:<expect>* that specifies the expected current value of the ref explicitly are still experimental and their semantics may change as we gain experience with this feature.

"--no-force-with-lease" will cancel all the previous --force-with-lease on the command line.

A general note on safety: supplying this option without an expected value, i.e. as *--force-with-lease* or *--force-with-lease=<refname>* interacts very badly with anything that implicitly runs *git fetch* on the remote to be pushed to in the background, e.g. *git fetch origin* on your repository in a cronjob.

The protection it offers over *--force* is ensuring that subsequent changes your work wasn't based on aren't clobbered, but this is trivially defeated if some background process is updating refs in the background. We don't have anything except the remote tracking info to go by as a heuristic for refs you're expected to have seen & are willing to clobber.

If your editor or some other system is running *git fetch* in the background for you a way to mitigate this is to simply set up another remote:

```
git remote add origin-push $(git config remote.origin.url)
git fetch origin-push
```

Now when the background process runs *git fetch origin* the references on *origin-push* won't be updated, and thus commands like:

```
git push --force-with-lease origin-push
```

Will fail unless you manually run *git fetch origin-push*. This method is of course entirely defeated by something that runs *git fetch --all*, in that case you'd need to either disable it or do something more tedious like:

```
git fetch               # update 'master' from remote
git tag base master     # mark our base point
git rebase -i master    # rewrite some commits
git push --force-with-lease=master:base master:master
```

I.e. create a *base* tag for versions of the upstream code that you've seen and are willing to overwrite, then rewrite history, and finally force push changes to *master* if the remote version is still at *base*, regardless of what your local *remotes/origin/master* has been updated to in the background.

Alternatively, specifying *--force-if-includes* as an ancillary option along with *--force-with-lease[=<refname>]* (i.e., without saying what exact commit the ref on the remote side must be pointing at, or which refs on the remote side are being protected) at the time of "push" will verify if updates from the remote-tracking refs that may have been implicitly updated in the background are integrated locally before allowing a forced update.

-f , --force

Usually, the command refuses to update a remote ref that is not an ancestor of the local ref used to overwrite it. Also, when *--force-with-lease* option is used, the command refuses to update a remote ref whose current value does not match what is expected.

This flag disables these checks, and can cause the remote repository to lose commits; use it with care.

Note that *--force* applies to all the refs that are pushed, hence using it with *push.default* set to *matching* or with multiple push destinations configured with *remote.*.push* may overwrite refs other than the current branch (including local refs that are strictly behind their remote counterpart). To force a push to only one branch, use a + in front of the refspec to push (e.g *git push origin +master* to force a push to the *master* branch). See the *<refspec>*... section above for details.

--[no-]force-if-includes

Force an update only if the tip of the remote-tracking ref has been integrated locally.

This option enables a check that verifies if the tip of the remote-tracking ref is reachable from one of the "reflog" entries of the local branch based in it for a rewrite. The check ensures that any updates from the remote have been incorporated locally by rejecting the forced update if that is not the case.

If the option is passed without specifying *--force-with-lease*, or specified along with *--force-with-lease=<ref-name>:<expect>*, it is a "no-op".

Specifying *--no-force-if-includes* disables this behavior.

--repo=<repository>

This option is equivalent to the <repository> argument. If both are specified, the command-line argument takes precedence.

-u , --set-upstream

For every branch that is up to date or successfully pushed, add upstream (tracking) reference, used by argument-less Section G.3.102, “git-pull(1)” and other commands. For more information, see *branch.<name>.merge* in Section G.3.29, “git-config(1)”.

--[no-]thin

These options are passed to Section G.3.125, “git-send-pack(1)”. A thin transfer significantly reduces the amount of sent data when the sender and receiver share many of the same objects in common. The default is *--thin*.

-q , --quiet

Suppress all output, including the listing of updated refs, unless an error occurs. Progress is not reported to the standard error stream.

-v , --verbose

Run verbosely.

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

--no-recurse-submodules , --recurse-submodules=check|on-demand|only|no

May be used to make sure all submodule commits used by the revisions to be pushed are available on a remote-tracking branch. If *check* is used Git will verify that all submodule commits that changed in the revisions to be pushed are available on at least one remote of the submodule. If any commits are missing the push will be aborted and exit with non-zero status. If *on-demand* is used all submodules that changed in the revisions to be pushed will be pushed. If on-demand was not able to push all necessary revisions it will also be aborted and exit with non-zero status. If *only* is used all submodules will be pushed while the superproject is left unpushed. A value of *no* or using *--no-recurse-submodules* can be used to override the push.recurseSubmodules configuration variable when no submodule recursion is required.

When using *on-demand* or *only*, if a submodule has a "push.recurseSubmodules={on-demand,only}" or "submodule.recurse" configuration, further recursion will occur. In this case, "only" is treated as "on-demand".

--[no-]verify

Toggle the pre-push hook (see Section G.4.7, “githooks(5)”). The default is --verify, giving the hook a chance to prevent the push. With --no-verify, the hook is bypassed completely.

-4 , --ipv4

Use IPv4 addresses only, ignoring IPv6 addresses.

-6 , --ipv6

Use IPv6 addresses only, ignoring IPv4 addresses.

GIT URLS

In general, URLs contain information about the transport protocol, the address of the remote server, and the path to the repository. Depending on the transport protocol, some of this information may be absent.

Git supports ssh, git, http, and https protocols (in addition, ftp and ftps can be used for fetching, but this is inefficient and deprecated; do not use them).

The native transport (i.e. git:// URL) does no authentication and should be used with caution on unsecured networks.

The following syntaxes may be used with them:

- `ssh://`[<*user*>`@`]<*host*>[`:`<*port*>]`/`<*path-to-git-repo*>
- `git://`<*host*>[:<*port*>]`/`<*path-to-git-repo*>
- `http[s]://`<*host*>[`:`<*port*>]`/`<*path-to-git-repo*>
- `ftp[s]://`<*host*>[`:`<*port*>]`/`<*path-to-git-repo*>

An alternative scp-like syntax may also be used with the ssh protocol:

- [<*user*>`@`]<*host*>`:/`<*path-to-git-repo*>

This syntax is only recognized if there are no slashes before the first colon. This helps differentiate a local path that contains a colon. For example the local path *foo:bar* could be specified as an absolute path or *./foo:bar* to avoid being misinterpreted as an ssh url.

The ssh and git protocols additionally support `~`<*username*> expansion:

- `ssh://`[<*user*>`@`]<*host*>[`:`<*port*>]`/~`<*user*>`/`<*path-to-git-repo*>
- `git://`<*host*>[`:`<*port*>]`/~`<*user*>`/`<*path-to-git-repo*>
- [<*user*>`@`]<*host*>`:~`<*user*>`/`<*path-to-git-repo*>

For local repositories, also supported by Git natively, the following syntaxes may be used:

- */path/to/repo.git/*
- `file:///path/to/repo.git/`

These two syntaxes are mostly equivalent, except when cloning, when the former implies *--local* option. See Section G.3.24, “git-clone(1)” for details.

git clone, *git fetch* and *git pull*, but not *git push*, will also accept a suitable bundle file. See Section G.3.12, “git-bundle(1)”.

When Git doesn't know how to handle a certain transport protocol, it attempts to use the *remote-*<*transport*> remote helper, if one exists. To explicitly request a remote helper, the following syntax may be used:

- <*transport*>::<*address*>

where <*address*> may be a path, a server and path, or an arbitrary URL-like string recognized by the specific remote helper being invoked. See Section G.4.12, “gitremote-helpers(7)” for details.

If there are a large number of similarly-named remote repositories and you want to use a different format for them (such that the URLs you use will be rewritten into URLs that work), you can create a configuration section of the form:

[url "<*actual-url-base*>"]
insteadOf = <*other-url-base*>

For example, with this:

```
[url "git://git.host.xz/"]
        insteadOf = host.xz:/path/to/
        insteadOf = work:
```

a URL like "work:repo.git" or like "host.xz:/path/to/repo.git" will be rewritten in any context that takes a URL to be "git://git.host.xz/repo.git".

If you want to rewrite URLs for push only, you can create a configuration section of the form:

[url "<*actual-url-base*>"]
pushInsteadOf = <*other-url-base*>

For example, with this:

```
[url "ssh://example.org/"]
        pushInsteadOf = git://example.org/
```

a URL like "git://example.org/path/to/repo.git" will be rewritten to "ssh://example.org/path/to/repo.git" for pushes, but pulls will still use the original URL.

REMOTES

The name of one of the following can be used instead of a URL as <*repository*> argument:

- a remote in the Git configuration file: *$GIT_DIR/config*,
- a file in the *$GIT_DIR/remotes* directory, or
- a file in the *$GIT_DIR/branches* directory.

All of these also allow you to omit the refspec from the command line because they each contain a refspec which git will use by default.

1. Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using Section G.3.112, “git-remote(1)”, Section G.3.29, “git-config(1)” or even by a manual edit to the *$GIT_DIR/config* file. The URL of this remote will be used to access the repository. The refspec of this remote will be used by default when you do not provide a refspec on the command line. The entry in the config file would appear like this:

```
[remote "<name>"]
        url = <URL>
        pushurl = <pushurl>
        push = <refspec>
        fetch = <refspec>
```

The <*pushurl*> is used for pushes only. It is optional and defaults to <*URL*>. Pushing to a remote affects all defined pushurls or all defined urls if no pushurls are defined. Fetch, however, will only fetch from the first defined url if multiple urls are defined.

2. Named file in *$GIT_DIR/remotes*

You can choose to provide the name of a file in *$GIT_DIR/remotes*. The URL in this file will be used to access the repository. The refspec in this file will be used as default when you do not provide a refspec on the command line. This file should have the following format:

```
URL: one of the above URL formats
Push: <refspec>
Pull: <refspec>
```

Push: lines are used by *git push* and *Pull:* lines are used by *git pull* and *git fetch*. Multiple *Push:* and *Pull:* lines may be specified for additional branch mappings.

3. Named file in *$GIT_DIR/branches*

You can choose to provide the name of a file in *$GIT_DIR/branches*. The URL in this file will be used to access the repository. This file should have the following format:

```
<URL>#<head>
```

<URL> is required; *#<head>* is optional.

Depending on the operation, git will use one of the following refspecs, if you don't provide one on the command line. *<branch>* is the name of this file in *$GIT_DIR/branches* and *<head>* defaults to *master*.

git fetch uses:

```
refs/heads/<head>:refs/heads/<branch>
```

git push uses:

```
HEAD:refs/heads/<head>
```

OUTPUT

The output of "git push" depends on the transport method used; this section describes the output when pushing over the Git protocol (either locally or via ssh).

The status of the push is output in tabular form, with each line representing the status of a single ref. Each line is of the form:

```
<flag> <summary> <from> -> <to> (<reason>)
```

If --porcelain is used, then each line of the output is of the form:

```
<flag> \t <from>:<to> \t <summary> (<reason>)
```

The status of up-to-date refs is shown only if --porcelain or --verbose option is used.

flag

A single character indicating the status of the ref:

(space)

for a successfully pushed fast-forward;

+

for a successful forced update;

-

for a successfully deleted ref;

*

for a successfully pushed new ref;

!

for a ref that was rejected or failed to push; and

=

for a ref that was up to date and did not need pushing.

summary

For a successfully pushed ref, the summary shows the old and new values of the ref in a form suitable for using as an argument to *git log* (this is *<old>..<new>* in most cases, and *<old>...<new>* for forced non-fast-forward updates).

For a failed update, more details are given:

rejected

Git did not try to send the ref at all, typically because it is not a fast-forward and you did not force the update.

remote rejected

The remote end refused the update. Usually caused by a hook on the remote side, or because the remote repository has one of the following safety options in effect: *receive.denyCurrentBranch* (for pushes to the checked out branch), *receive.denyNonFastForwards* (for forced non-fast-forward updates), *receive.denyDeletes* or *receive.denyDeleteCurrent*. See Section G.3.29, “git-config(1)”.

remote failure

The remote end did not report the successful update of the ref, perhaps because of a temporary error on the remote side, a break in the network connection, or other transient error.

from

The name of the local ref being pushed, minus its *refs/<type>/* prefix. In the case of deletion, the name of the local ref is omitted.

to

The name of the remote ref being updated, minus its *refs/<type>/* prefix.

reason

A human-readable explanation. In the case of successfully pushed refs, no explanation is needed. For a failed ref, the reason for failure is described.

NOTE ABOUT FAST-FORWARDS

When an update changes a branch (or more in general, a ref) that used to point at commit A to point at another commit B, it is called a fast-forward update if and only if B is a descendant of A.

In a fast-forward update from A to B, the set of commits that the original commit A built on top of is a subset of the commits the new commit B builds on top of. Hence, it does not lose any history.

In contrast, a non-fast-forward update will lose history. For example, suppose you and somebody else started at the same commit X, and you built a history leading to commit B while the other person built a history leading to commit A. The history looks like this:

```
      B
     /
```

```
---X---A
```

Further suppose that the other person already pushed changes leading to A back to the original repository from which you two obtained the original commit X.

The push done by the other person updated the branch that used to point at commit X to point at commit A. It is a fast-forward.

But if you try to push, you will attempt to update the branch (that now points at A) with commit B. This does *not* fast-forward. If you did so, the changes introduced by commit A will be lost, because everybody will now start building on top of B.

The command by default does not allow an update that is not a fast-forward to prevent such loss of history.

If you do not want to lose your work (history from X to B) or the work by the other person (history from X to A), you would need to first fetch the history from the repository, create a history that contains changes done by both parties, and push the result back.

You can perform "git pull", resolve potential conflicts, and "git push" the result. A "git pull" will create a merge commit C between commits A and B.

```
      B---C
     /   /
---X---A
```

Updating A with the resulting merge commit will fast-forward and your push will be accepted.

Alternatively, you can rebase your change between X and B on top of A, with "git pull --rebase", and push the result back. The rebase will create a new commit D that builds the change between X and B on top of A.

```
      B   D
     /   /
---X---A
```

Again, updating A with this commit will fast-forward and your push will be accepted.

There is another common situation where you may encounter non-fast-forward rejection when you try to push, and it is possible even when you are pushing into a repository nobody else pushes into. After you push commit A yourself (in the first picture in this section), replace it with "git commit --amend" to produce commit B, and you try to push it out, because forgot that you have pushed A out already. In such a case, and only if you are certain that nobody in the meantime fetched your earlier commit A (and started building on top of it), you can run "git push --force" to overwrite it. In other words, "git push --force" is a method reserved for a case where you do mean to lose history.

EXAMPLES

git push

Works like *git push <remote>*, where <remote> is the current branch's remote (or *origin*, if no remote is configured for the current branch).

git push origin

Without additional configuration, pushes the current branch to the configured upstream (*branch.<name>.merge* configuration variable) if it has the same name as the current branch, and errors out without pushing otherwise.

The default behavior of this command when no <refspec> is given can be configured by setting the *push* option of the remote, or the *push.default* configuration variable.

For example, to default to pushing only the current branch to *origin* use *git config remote.origin.push HEAD*. Any valid <refspec> (like the ones in the examples below) can be configured as the default for *git push origin*.

git push origin :

Push "matching" branches to *origin*. See <refspec> in the OPTIONS section above for a description of "matching" branches.

git push origin master

Find a ref that matches *master* in the source repository (most likely, it would find *refs/heads/master*), and update the same ref (e.g. *refs/heads/master*) in *origin* repository with it. If *master* did not exist remotely, it would be created.

git push origin HEAD

A handy way to push the current branch to the same name on the remote.

git push mothership master:satellite/master dev:satellite/dev

Use the source ref that matches *master* (e.g. *refs/heads/master*) to update the ref that matches *satellite/master* (most probably *refs/remotes/satellite/master*) in the *mothership* repository; do the same for *dev* and *satellite/dev*.

See the section describing *<refspec>*... above for a discussion of the matching semantics.

This is to emulate *git fetch* run on the *mothership* using *git push* that is run in the opposite direction in order to integrate the work done on *satellite*, and is often necessary when you can only make connection in one way (i.e. satellite can ssh into mothership but mothership cannot initiate connection to satellite because the latter is behind a firewall or does not run sshd).

After running this *git push* on the *satellite* machine, you would ssh into the *mothership* and run *git merge* there to complete the emulation of *git pull* that were run on *mothership* to pull changes made on *satellite*.

git push origin HEAD:master

Push the current branch to the remote ref matching *master* in the *origin* repository. This form is convenient to push the current branch without thinking about its local name.

git push origin master:refs/heads/experimental

Create the branch *experimental* in the *origin* repository by copying the current *master* branch. This form is only needed to create a new branch or tag in the remote repository when the local name and the remote name are different; otherwise, the ref name on its own will work.

git push origin :experimental

Find a ref that matches *experimental* in the *origin* repository (e.g. *refs/heads/experimental*), and delete it.

git push origin +dev:master

Update the origin repository's master branch with the dev branch, allowing non-fast-forward updates. **This can leave unreferenced commits dangling in the origin repository.** Consider the following situation, where a fast-forward is not possible:

```
o---o---o---A---B  origin/master
          \
           X---Y---Z  dev
```

The above command would change the origin repository to

```
           A---B  (unnamed branch)
          /
o---o---o---X---Y---Z  master
```

Commits A and B would no longer belong to a branch with a symbolic name, and so would be unreachable. As such, these commits would be removed by a *git gc* command on the origin repository.

SECURITY

The fetch and push protocols are not designed to prevent one side from stealing data from the other repository that was not intended to be shared. If you have private data that you need to protect from a malicious peer, your best option is to store it in another repository. This applies to both clients and servers. In particular, namespaces on a server are not effective for read access control; you should only grant read access to a namespace to clients that you would trust with read access to the entire repository.

The known attack vectors are as follows:

1. The victim sends "have" lines advertising the IDs of objects it has that are not explicitly intended to be shared but can be used to optimize the transfer if the peer also has them. The attacker chooses an object ID X to steal and sends a ref to X, but isn't required to send the content of X because the victim already has it. Now the victim believes that the attacker has X, and it sends the content of X back to the attacker later. (This attack is most straightforward for a client to perform on a server, by creating a ref to X in the namespace the client has access to and then fetching it. The most likely way for a server to perform it on a client is to "merge" X into a public branch and hope that the user does additional work on this branch and pushes it back to the server without noticing the merge.)

2. As in #1, the attacker chooses an object ID X to steal. The victim sends an object Y that the attacker already has, and the attacker falsely claims to have X and not Y, so the victim sends Y as a delta against X. The delta reveals regions of X that are similar to Y to the attacker.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

push.autoSetupRemote

If set to "true" assume *--set-upstream* on default push when no upstream tracking exists for the current branch; this option takes effect with push.default options *simple*, *upstream*, and *current*. It is useful if by default you want new branches to be pushed to the default remote (like the behavior of *push.default=current*) and you also want the upstream tracking to be set. Workflows most likely to benefit from this option are *simple* central workflows where all branches are expected to have the same name on the remote.

push.default

Defines the action *git push* should take if no refspec is given (whether from the command-line, config, or elsewhere). Different values are well-suited for specific workflows; for instance, in a purely central workflow (i.e. the fetch source is equal to the push destination), *upstream* is probably what you want. Possible values are:

- *nothing* - do not push anything (error out) unless a refspec is given. This is primarily meant for people who want to avoid mistakes by always being explicit.

- *current* - push the current branch to update a branch with the same name on the receiving end. Works in both central and non-central workflows.

- *upstream* - push the current branch back to the branch whose changes are usually integrated into the current branch (which is called *@{upstream}*). This mode only makes sense if you are pushing to the same repository you would normally pull from (i.e. central workflow).

- *tracking* - This is a deprecated synonym for *upstream*.

- *simple* - push the current branch with the same name on the remote.

  If you are working on a centralized workflow (pushing to the same repository you pull from, which is typically *origin*), then you need to configure an upstream branch with the same name.

This mode is the default since Git 2.0, and is the safest option suited for beginners.

- *matching* - push all branches having the same name on both ends. This makes the repository you are pushing to remember the set of branches that will be pushed out (e.g. if you always push *maint* and *master* there and no other branches, the repository you push to will have these two branches, and your local *maint* and *master* will be pushed there).

  To use this mode effectively, you have to make sure *all* the branches you would push out are ready to be pushed out before running *git push*, as the whole point of this mode is to allow you to push all of the branches in one go. If you usually finish work on only one branch and push out the result, while other branches are unfinished, this mode is not for you. Also this mode is not suitable for pushing into a shared central repository, as other people may add new branches there, or update the tip of existing branches outside your control.

  This used to be the default, but not since Git 2.0 (*simple* is the new default).

push.followTags

If set to true, enable *--follow-tags* option by default. You may override this configuration at time of push by specifying *--no-follow-tags*.

push.gpgSign

May be set to a boolean value, or the string *if-asked*. A true value causes all pushes to be GPG signed, as if *--signed* is passed to Section G.3.103, “git-push(1)”. The string *if-asked* causes pushes to be signed if the server supports it, as if *--signed=if-asked* is passed to *git push*. A false value may override a value from a lower-priority config file. An explicit command-line flag always overrides this config option.

push.pushOption

When no *--push-option=<option>* argument is given from the command line, *git push* behaves as if each <value> of this variable is given as *--push-option=<value>*.

This is a multi-valued variable, and an empty value can be used in a higher priority configuration file (e.g. *.git/config* in a repository) to clear the values inherited from a lower priority configuration files (e.g. *$HOME/.gitconfig*).

```
Example:

/etc/gitconfig
  push.pushoption = a
  push.pushoption = b

~/.gitconfig
  push.pushoption = c

repo/.git/config
  push.pushoption =
  push.pushoption = b

This will result in only b (a and c are cleared).
```

push.recurseSubmodules

May be "check", "on-demand", "only", or "no", with the same behavior as that of "push --recurse-submodules". If not set, *no* is used by default, unless *submodule.recurse* is set (in which case a *true* value means *on-demand*).

push.useForceIfIncludes

If set to "true", it is equivalent to specifying *--force-if-includes* as an option to Section G.3.103, “git-push(1)” in the command line. Adding *--no-force-if-includes* at the time of push overrides this configuration setting.

push.negotiate

If set to "true", attempt to reduce the size of the packfile sent by rounds of negotiation in which the client and the server attempt to find commits in common. If "false", Git will rely solely on the server's ref advertisement to find commits in common.

push.useBitmaps

If set to "false", disable use of bitmaps for "git push" even if *pack.useBitmaps* is "true", without preventing other git operations from using bitmaps. Default is true.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.104. git-quiltimport(1)

2024-07-02
Revision History
2024-07-02

NAME

git-quiltimport - Applies a quilt patchset onto the current branch

SYNOPSIS

git quiltimport [--dry-run | -n] [--author <author>] [--patches <dir>]
[--series <file>] [--keep-non-patch]

DESCRIPTION

Applies a quilt patchset onto the current Git branch, preserving the patch boundaries, patch order, and patch descriptions present in the quilt patchset.

For each patch the code attempts to extract the author from the patch description. If that fails it falls back to the author specified with --author. If the --author flag was not given the patch description is displayed and the user is asked to interactively enter the author of the patch.

If a subject is not found in the patch description the patch name is preserved as the 1 line subject in the Git description.

OPTIONS

-n , --dry-run

Walk through the patches in the series and warn if we cannot find all of the necessary information to commit a patch. At the time of this writing only missing author information is warned about.

--author *Author Name <Author Email>*

The author name and email address to use when no author information can be found in the patch description.

--patches <dir>

The directory to find the quilt patches.

The default for the patch directory is *patches* or the value of the *$QUILT_PATCHES* environment variable.

--series <file>

The quilt series file.

The default for the series file is <patches>/series or the value of the *$QUILT_SERIES* environment variable.

--keep-non-patch

Pass *-b* flag to *git mailinfo* (see Section G.3.78, “git-mailinfo(1)”).

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.105. git-range-diff(1)

2024-07-02
Revision History
2024-07-02

NAME

git-range-diff - Compare two commit ranges (e.g. two versions of a branch)

SYNOPSIS

git range-diff [--color=[<when>]] [--no-color] [<diff-options>]
[--no-dual-color] [--creation-factor=<factor>]
[--left-only | --right-only]
(<range1> <range2> | <rev1>…<rev2> | <base> <rev1> <rev2>)
[[--] <path>…]

DESCRIPTION

This command shows the differences between two versions of a patch series, or more generally, two commit ranges (ignoring merge commits).

In the presence of *<path>* arguments, these commit ranges are limited accordingly.

To that end, it first finds pairs of commits from both commit ranges that correspond with each other. Two commits are said to correspond when the diff between their patches (i.e. the author information, the commit message and the commit diff) is reasonably small compared to the patches' size. See ``Algorithm`` below for details.

Finally, the list of matching commits is shown in the order of the second commit range, with unmatched commits being inserted just after all of their ancestors have been shown.

There are three ways to specify the commit ranges:

- *<range1> <range2>*: Either commit range can be of the form *<base>..<rev>*, *<rev>^!* or *<rev>^-<n>*. See *SPECIFYING RANGES* in Section G.4.14, “gitrevisions(7)” for more details.
- *<rev1>...<rev2>*. This is equivalent to *<rev2>..<rev1> <rev1>..<rev2>*.
- *<base> <rev1> <rev2>*: This is equivalent to *<base>..<rev1> <base>..<rev2>*.

OPTIONS

--no-dual-color

When the commit diffs differ, git range-diff` recreates the original diffs coloring, and adds outer -/+ diff markers with the **background** being red/green to make it easier to see e.g. when there was a change in what exact lines were added.

Additionally, the commit diff lines that are only present in the first commit range are shown "dimmed" (this can be overridden using the *color.diff.<slot>* config setting where *<slot>* is one of *contextDimmed*, *oldDimmed* and *newDimmed*), and the commit diff lines that are only present in the second commit range are shown in bold

(which can be overridden using the config settings *color.diff.<slot>* with *<slot>* being one of *contextBold*, *oldBold* or *newBold*).

This is known to *range-diff* as "dual coloring". Use *--no-dual-color* to revert to color all lines according to the outer diff markers (and completely ignore the inner diff when it comes to color).

--creation-factor=<percent>

Set the creation/deletion cost fudge factor to *<percent>*. Defaults to 60. Try a larger value if *git range-diff* erroneously considers a large change a total rewrite (deletion of one commit and addition of another), and a smaller one in the reverse case. See the ``Algorithm`` section below for an explanation of why this is needed.

--left-only

Suppress commits that are missing from the first specified range (or the "left range" when using the *<rev1>...<rev2>* format).

--right-only

Suppress commits that are missing from the second specified range (or the "right range" when using the *<rev1>...<rev2>* format).

--[no-]notes[=<ref>]

This flag is passed to the *git log* program (see Section G.3.74, “git-log(1)”) that generates the patches.

<range1> <range2>

Compare the commits specified by the two ranges, where *<range1>* is considered an older version of *<range2>*.

<rev1>…<rev2>

Equivalent to passing *<rev2>..<rev1>* and *<rev1>..<rev2>*.

<base> <rev1> <rev2>

Equivalent to passing *<base>..<rev1>* and *<base>..<rev2>*. Note that *<base>* does not need to be the exact branch point of the branches. Example: after rebasing a branch *my-topic*, *git range-diff my-topic@{u} my-topic@{1} my-topic* would show the differences introduced by the rebase.

git range-diff also accepts the regular diff options (see Section G.3.44, “git-diff(1)”), most notably the *--color=[<when>]* and *--no-color* options. These options are used when generating the "diff between patches", i.e. to compare the author, commit message and diff of corresponding old/new commits. There is currently no means to tweak most of the diff options passed to *git log* when generating those patches.

OUTPUT STABILITY

The output of the *range-diff* command is subject to change. It is intended to be human-readable porcelain output, not something that can be used across versions of Git to get a textually stable *range-diff* (as opposed to something like the *--stable* option to Section G.3.99, “git-patch-id(1)”). There's also no equivalent of Section G.3.5, “git-apply(1)” for *range-diff*, the output is not intended to be machine-readable.

This is particularly true when passing in diff options. Currently some options like *--stat* can, as an emergent effect, produce output that's quite useless in the context of *range-diff*. Future versions of *range-diff* may learn to interpret such options in a manner specific to *range-diff* (e.g. for *--stat* producing human-readable output which summarizes how the diffstat changed).

CONFIGURATION

This command uses the *diff.color.** and *pager.range-diff* settings (the latter is on by default). See Section G.3.29, “git-config(1)”.

EXAMPLES

When a rebase required merge conflicts to be resolved, compare the changes introduced by the rebase directly afterwards using:

```
$ git range-diff @{u} @{1} @
```

A typical output of *git range-diff* would look like this:

```
-:  ------- > 1:  0ddba11 Prepare for the inevitable!
1:  c0debee = 2:  cab005e Add a helpful message at the start
2:  f00dbal ! 3:  decafe1 Describe a bug
    @@ -1,3 +1,3 @@
     Author: A U Thor <author@example.com>

    -TODO: Describe a bug
    +Describe a bug
    @@ -324,5 +324,6
      This is expected.

    -+What is unexpected is that it will also crash.
    ++Unexpectedly, it also crashes. This is a bug, and the jury is
    ++still out there how to fix it best. See ticket #314 for details.

      Contact
3:  bedead < -:  ------- TO-UNDO
```

In this example, there are 3 old and 3 new commits, where the developer removed the 3rd, added a new one before the first two, and modified the commit message of the 2nd commit as well as its diff.

When the output goes to a terminal, it is color-coded by default, just like regular *git diff*'s output. In addition, the first line (adding a commit) is green, the last line (deleting a commit) is red, the second line (with a perfect match) is yellow like the commit header of *git show*'s output, and the third line colors the old commit red, the new one green and the rest like *git show*'s commit header.

A naive color-coded diff of diffs is actually a bit hard to read, though, as it colors the entire lines red or green. The line that added "What is unexpected" in the old commit, for example, is completely red, even if the intent of the old commit was to add something.

To help with that, *range* uses the *--dual-color* mode by default. In this mode, the diff of diffs will retain the original diff colors, and prefix the lines with -/+ markers that have their **background** red or green, to make it more obvious that they describe how the diff itself changed.

Algorithm

The general idea is this: we generate a cost matrix between the commits in both commit ranges, then solve the least-cost assignment.

The cost matrix is populated thusly: for each pair of commits, both diffs are generated and the "diff of diffs" is generated, with 3 context lines, then the number of lines in that diff is used as cost.

To avoid false positives (e.g. when a patch has been removed, and an unrelated patch has been added between two iterations of the same patch series), the cost matrix is extended to allow for that, by adding fixed-cost entries for wholesale deletes/adds.

Example: Let commits *1--2* be the first iteration of a patch series and *A--C* the second iteration. Let's assume that *A* is a cherry-pick of *2*, and *C* is a cherry-pick of *1* but with a small modification (say, a fixed typo). Visualize the commits as a bipartite graph:

```
    1             A
```

```
2             B

              C
```

We are looking for a "best" explanation of the new series in terms of the old one. We can represent an "explanation" as an edge in the graph:

```
1             A
            /
2 --------'   B

              C
```

This explanation comes for "free" because there was no change. Similarly *C* could be explained using *1*, but that comes at some cost c>0 because of the modification:

```
1 ----.       A
      |     /
2 ----+---'   B
      |
      `----- C
      c>0
```

In mathematical terms, what we are looking for is some sort of a minimum cost bipartite matching; 1` is matched to *C* at some cost, etc. The underlying graph is in fact a complete bipartite graph; the cost we associate with every edge is the size of the diff between the two commits patches. To explain also new commits, we introduce dummy nodes on both sides:

```
1 ----.       A
      |     /
2 ----+---'   B
      |
o     `----- C
      c>0
o             o

o             o
```

The cost of an edge *o--C* is the size of *C*'s diff, modified by a fudge factor that should be smaller than 100%. The cost of an edge *o--o* is free. The fudge factor is necessary because even if *1* and *C* have nothing in common, they may still share a few empty lines and such, possibly making the assignment *1--C*, *o--o* slightly cheaper than *1--o*, *o--C* even if *1* and *C* have nothing in common. With the fudge factor we require a much larger common part to consider patches as corresponding.

The overall time needed to compute this algorithm is the time needed to compute n+m commit diffs and then n*m diffs of patches, plus the time needed to compute the least-cost assignment between n and m diffs. Git uses an implementation of the Jonker-Volgenant algorithm to solve the assignment problem, which has cubic runtime complexity. The matching found in this case will look like this:

```
1 ----.       A
      |     /
2 ----+---'   B
   .--+-----'
o -'  `----- C
      c>0
o ---------- o

o ---------- o
```

SEE ALSO

Section G.3.74, “git-log(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.106. git-read-tree(1)

2024-07-02
Revision History
2024-07-02

NAME

git-read-tree - Reads tree information into the index

SYNOPSIS

git read-tree [(-m [--trivial] [--aggressive] | --reset | --prefix=<prefix>)
[-u | -i]] [--index-output=<file>] [--no-sparse-checkout]
(--empty | <tree-ish1> [<tree-ish2> [<tree-ish3>]])

DESCRIPTION

Reads the tree information given by <tree-ish> into the index, but does not actually **update** any of the files it "caches". (see: Section G.3.18, “git-checkout-index(1)”)

Optionally, it can merge a tree into the index, perform a fast-forward (i.e. 2-way) merge, or a 3-way merge, with the *-m* flag. When used with *-m*, the *-u* flag causes it to also update the files in the work tree with the result of the merge.

Only trivial merges are done by *git read-tree* itself. Only conflicting paths will be in an unmerged state when *git read-tree* returns.

OPTIONS

-m

Perform a merge, not just a read. The command will refuse to run if your index file has unmerged entries, indicating that you have not finished a previous merge you started.

--reset

Same as -m, except that unmerged entries are discarded instead of failing. When used with *-u*, updates leading to loss of working tree changes or untracked files or directories will not abort the operation.

-u

After a successful merge, update the files in the work tree with the result of the merge.

-i

Usually a merge requires the index file as well as the files in the working tree to be up to date with the current head commit, in order not to lose local changes. This flag disables the check with the working tree and is meant to be used when creating a merge of trees that are not directly related to the current working tree status into a temporary index file.

-n , --dry-run

Check if the command would error out, without updating the index or the files in the working tree for real.

-v

Show the progress of checking files out.

--trivial

Restrict three-way merge by *git read-tree* to happen only if there is no file-level merging required, instead of resolving merge for trivial cases and leaving conflicting files unresolved in the index.

--aggressive

Usually a three-way merge by *git read-tree* resolves the merge for really trivial cases and leaves other cases unresolved in the index, so that porcelains can implement different merge policies. This flag makes the command resolve a few more cases internally:

- when one side removes a path and the other side leaves the path unmodified. The resolution is to remove that path.
- when both sides remove a path. The resolution is to remove that path.
- when both sides add a path identically. The resolution is to add that path.

--prefix=<prefix>

Keep the current index contents, and read the contents of the named tree-ish under the directory at *<prefix>*. The command will refuse to overwrite entries that already existed in the original index file.

--index-output=<file>

Instead of writing the results out to *$GIT_INDEX_FILE*, write the resulting index in the named file. While the command is operating, the original index file is locked with the same mechanism as usual. The file must allow to be rename(2)ed into from a temporary file that is created next to the usual index file; typically this means it needs to be on the same filesystem as the index file itself, and you need write permission to the directories the index file and index output file are located in.

--[no-]recurse-submodules

Using --recurse-submodules will update the content of all active submodules according to the commit recorded in the superproject by calling read-tree recursively, also setting the submodules' HEAD to be detached at that commit.

--no-sparse-checkout

Disable sparse checkout support even if *core.sparseCheckout* is true.

--empty

Instead of reading tree object(s) into the index, just empty it.

-q , --quiet

Quiet, suppress feedback messages.

<tree-ish#>

The id of the tree object(s) to be read/merged.

MERGING

If *-m* is specified, *git read-tree* can perform 3 kinds of merge, a single tree merge if only 1 tree is given, a fast-forward merge with 2 trees, or a 3-way merge if 3 or more trees are provided.

1. Single Tree Merge

If only 1 tree is specified, *git read-tree* operates as if the user did not specify *-m*, except that if the original index has an entry for a given pathname, and the contents of the path match with the tree being read, the stat info from the index is used. (In other words, the index's stat()s take precedence over the merged tree's).

That means that if you do a *git read-tree -m <newtree>* followed by a *git checkout-index -f -u -a*, the *git checkout-index* only checks out the stuff that really changed.

This is used to avoid unnecessary false hits when *git diff-files* is run after *git read-tree*.

2. Two Tree Merge

Typically, this is invoked as *git read-tree -m $H $M*, where $H is the head commit of the current repository, and $M is the head of a foreign tree, which is simply ahead of $H (i.e. we are in a fast-forward situation).

When two trees are specified, the user is telling *git read-tree* the following:

1. The current index and work tree is derived from $H, but the user may have local changes in them since $H.

2. The user wants to fast-forward to $M.

In this case, the *git read-tree -m $H $M* command makes sure that no local change is lost as the result of this "merge". Here are the "carry forward" rules, where "I" denotes the index, "clean" means that index and work tree coincide, and "exists"/"nothing" refer to the presence of a path in the specified commit:

```
   I                   H        M        Result
  -------------------------------------------------------
0  nothing             nothing  nothing  (does not happen)
1  nothing             nothing  exists   use M
2  nothing             exists   nothing  remove path from index
3  nothing             exists   exists,  use M if "initial checkout",
                                H == M   keep index otherwise
                                exists,  fail
                                H != M

   clean I==H  I==M
  ------------------
4  yes   N/A   N/A     nothing  nothing  keep index
5  no    N/A   N/A     nothing  nothing  keep index

6  yes   N/A   yes     nothing  exists   keep index
7  no    N/A   yes     nothing  exists   keep index
8  yes   N/A   no      nothing  exists   fail
9  no    N/A   no      nothing  exists   fail

10 yes   yes   N/A     exists   nothing  remove path from index
11 no    yes   N/A     exists   nothing  fail
12 yes   no    N/A     exists   nothing  fail
13 no    no    N/A     exists   nothing  fail

   clean (H==M)
  ------
14 yes                 exists   exists   keep index
15 no                  exists   exists   keep index

   clean I==H  I==M (H!=M)
  ------------------
16 yes   no    no      exists   exists   fail
```

```
17 no    no    no      exists   exists   fail
18 yes   no    yes     exists   exists   keep index
19 no    no    yes     exists   exists   keep index
20 yes   yes   no      exists   exists   use M
21 no    yes   no      exists   exists   fail
```

In all "keep index" cases, the index entry stays as in the original index file. If the entry is not up to date, *git read-tree* keeps the copy in the work tree intact when operating under the -u flag.

When this form of *git read-tree* returns successfully, you can see which of the "local changes" that you made were carried forward by running *git diff-index --cached $M*. Note that this does not necessarily match what *git diff-index --cached $H* would have produced before such a two tree merge. This is because of cases 18 and 19 -- if you already had the changes in $M (e.g. maybe you picked it up via e-mail in a patch form), *git diff-index --cached $H* would have told you about the change before this merge, but it would not show in *git diff-index --cached $M* output after the two-tree merge.

Case 3 is slightly tricky and needs explanation. The result from this rule logically should be to remove the path if the user staged the removal of the path and then switching to a new branch. That however will prevent the initial checkout from happening, so the rule is modified to use M (new tree) only when the content of the index is empty. Otherwise the removal of the path is kept as long as $H and $M are the same.

3. 3-Way Merge

Each "index" entry has two bits worth of "stage" state. stage 0 is the normal one, and is the only one you'd see in any kind of normal use.

However, when you do *git read-tree* with three trees, the "stage" starts out at 1.

This means that you can do

```
$ git read-tree -m <tree1> <tree2> <tree3>
```

and you will end up with an index with all of the <tree1> entries in "stage1", all of the <tree2> entries in "stage2" and all of the <tree3> entries in "stage3". When performing a merge of another branch into the current branch, we use the common ancestor tree as <tree1>, the current branch head as <tree2>, and the other branch head as <tree3>.

Furthermore, *git read-tree* has special-case logic that says: if you see a file that matches in all respects in the following states, it "collapses" back to "stage0":

- stage 2 and 3 are the same; take one or the other (it makes no difference - the same work has been done on our branch in stage 2 and their branch in stage 3)
- stage 1 and stage 2 are the same and stage 3 is different; take stage 3 (our branch in stage 2 did not do anything since the ancestor in stage 1 while their branch in stage 3 worked on it)
- stage 1 and stage 3 are the same and stage 2 is different take stage 2 (we did something while they did nothing)

The *git write-tree* command refuses to write a nonsensical tree, and it will complain about unmerged entries if it sees a single entry that is not stage 0.

OK, this all sounds like a collection of totally nonsensical rules, but it's actually exactly what you want in order to do a fast merge. The different stages represent the "result tree" (stage 0, aka "merged"), the original tree (stage 1, aka "orig"), and the two trees you are trying to merge (stage 2 and 3 respectively).

The order of stages 1, 2 and 3 (hence the order of three <tree-ish> command-line arguments) are significant when you start a 3-way merge with an index file that is already populated. Here is an outline of how the algorithm works:

- if a file exists in identical format in all three trees, it will automatically collapse to "merged" state by *git read-tree*.
- a file that has *any* difference what-so-ever in the three trees will stay as separate entries in the index. It's up to "porcelain policy" to determine how to remove the non-0 stages, and insert a merged version.

- the index file saves and restores with all this information, so you can merge things incrementally, but as long as it has entries in stages 1/2/3 (i.e., "unmerged entries") you can't write the result. So now the merge algorithm ends up being really simple:
  - you walk the index in order, and ignore all entries of stage 0, since they've already been done.
  - if you find a "stage1", but no matching "stage2" or "stage3", you know it's been removed from both trees (it only existed in the original tree), and you remove that entry.
  - if you find a matching "stage2" and "stage3" tree, you remove one of them, and turn the other into a "stage0" entry. Remove any matching "stage1" entry if it exists too. .. all the normal trivial rules ..

You would normally use *git merge-index* with supplied *git merge-one-file* to do this last step. The script updates the files in the working tree as it merges each path and at the end of a successful merge.

When you start a 3-way merge with an index file that is already populated, it is assumed that it represents the state of the files in your work tree, and you can even have files with changes unrecorded in the index file. It is further assumed that this state is "derived" from the stage 2 tree. The 3-way merge refuses to run if it finds an entry in the original index file that does not match stage 2.

This is done to prevent you from losing your work-in-progress changes, and mixing your random changes in an unrelated merge commit. To illustrate, suppose you start from what has been committed last to your repository:

```
$ JC=`git rev-parse --verify "HEAD^0"`
$ git checkout-index -f -u -a $JC
```

You do random edits, without running *git update-index*. And then you notice that the tip of your "upstream" tree has advanced since you pulled from him:

```
$ git fetch git://.... linus
$ LT=`git rev-parse FETCH_HEAD`
```

Your work tree is still based on your HEAD ($JC), but you have some edits since. Three-way merge makes sure that you have not added or modified index entries since $JC, and if you haven't, then does the right thing. So with the following sequence:

```
$ git read-tree -m -u `git merge-base $JC $LT` $JC $LT
$ git merge-index git-merge-one-file -a
$ echo "Merge with Linus" | \
  git commit-tree `git write-tree` -p $JC -p $LT
```

what you would commit is a pure merge between $JC and $LT without your work-in-progress changes, and your work tree would be updated to the result of the merge.

However, if you have local changes in the working tree that would be overwritten by this merge, *git read-tree* will refuse to run to prevent your changes from being lost.

In other words, there is no need to worry about what exists only in the working tree. When you have local changes in a part of the project that is not involved in the merge, your changes do not interfere with the merge, and are kept intact. When they **do** interfere, the merge does not even start (*git read-tree* complains loudly and fails without modifying anything). In such a case, you can simply continue doing what you were in the middle of doing, and when your working tree is ready (i.e. you have finished your work-in-progress), attempt the merge again.

SPARSE CHECKOUT

Note: The skip-worktree capabilities in Section G.3.147, “git-update-index(1)” and *read-tree* predated the introduction of Section G.3.135, “git-sparse-checkout(1)”. Users are encouraged to use the *sparse-checkout* command in preference to these plumbing commands for sparse-checkout/skip-worktree related needs. However, the information below might be useful to users trying to understand the pattern style used in non-cone mode of the *sparse-checkout* command.

"Sparse checkout" allows populating the working directory sparsely. It uses the skip-worktree bit (see Section G.3.147, “git-update-index(1)”) to tell Git whether a file in the working directory is worth looking at.

git read-tree and other merge-based commands (*git merge*, *git checkout*…) can help maintaining the skip-worktree bitmap and working directory update. *$GIT_DIR/info/sparse-checkout* is used to define the skip-worktree reference bitmap. When *git read-tree* needs to update the working directory, it resets the skip-worktree bit in the index based on this file, which uses the same syntax as .gitignore files. If an entry matches a pattern in this file, or the entry corresponds to a file present in the working tree, then skip-worktree will not be set on that entry. Otherwise, skip-worktree will be set.

Then it compares the new skip-worktree value with the previous one. If skip-worktree turns from set to unset, it will add the corresponding file back. If it turns from unset to set, that file will be removed.

While *$GIT_DIR/info/sparse-checkout* is usually used to specify what files are in, you can also specify what files are *not* in, using negate patterns. For example, to remove the file *unwanted*:

```
/*
!unwanted
```

Another tricky thing is fully repopulating the working directory when you no longer want sparse checkout. You cannot just disable "sparse checkout" because skip-worktree bits are still in the index and your working directory is still sparsely populated. You should re-populate the working directory with the *$GIT_DIR/info/sparse-checkout* file content as follows:

```
/*
```

Then you can disable sparse checkout. Sparse checkout support in *git read-tree* and similar commands is disabled by default. You need to turn *core.sparseCheckout* on in order to have sparse checkout support.

SEE ALSO

Section G.3.160, “git-write-tree(1)”, Section G.3.75, “git-ls-files(1)”, Section G.4.5, “gitignore(5)”, Section G.3.135, “git-sparse-checkout(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.107. git-rebase(1)

2024-07-02
Revision History
2024-07-02

NAME

git-rebase - Reapply commits on top of another base tip

SYNOPSIS

git rebase [-i | --interactive] [<options>] [--exec <cmd>]
[--onto <newbase> | --keep-base] [<upstream> [<branch>]]
git rebase [-i | --interactive] [<options>] [--exec <cmd>] [--onto <newbase>]
--root [<branch>]
git rebase (--continue|--skip|--abort|--quit|--edit-todo|--show-current-patch)

DESCRIPTION

If *<branch>* is specified, *git rebase* will perform an automatic *git switch <branch>* before doing anything else. Otherwise it remains on the current branch.

If *<upstream>* is not specified, the upstream configured in *branch.<name>.remote* and *branch.<name>.merge* options will be used (see Section G.3.29, “git-config(1)” for details) and the *--fork-point* option is assumed. If you are currently not on any branch or if the current branch does not have a configured upstream, the rebase will abort.

All changes made by commits in the current branch but that are not in *<upstream>* are saved to a temporary area. This is the same set of commits that would be shown by *git log <upstream>..HEAD*; or by *git log 'fork_point'..HEAD*, if *--fork-point* is active (see the description on *--fork-point* below); or by *git log HEAD*, if the *--root* option is specified.

The current branch is reset to *<upstream>* or *<newbase>* if the *--onto* option was supplied. This has the exact same effect as *git reset --hard <upstream>* (or *<newbase>*). *ORIG_HEAD* is set to point at the tip of the branch before the reset.

> **Note**
>
> *ORIG_HEAD* is not guaranteed to still point to the previous branch tip at the end of the rebase if other commands that write that pseudo-ref (e.g. *git reset*) are used during the rebase. The previous branch tip, however, is accessible using the reflog of the current branch (i.e. *@{1}*, see Section G.4.14, “gitrevisions(7)”).

The commits that were previously saved into the temporary area are then reapplied to the current branch, one by one, in order. Note that any commits in *HEAD* which introduce the same textual changes as a commit in *HEAD..<upstream>* are omitted (i.e., a patch already accepted upstream with a different commit message or timestamp will be skipped).

It is possible that a merge failure will prevent this process from being completely automatic. You will have to resolve any such merge failure and run *git rebase --continue*. Another option is to bypass the commit that caused the merge failure with *git rebase --skip*. To check out the original *<branch>* and remove the *.git/rebase-apply* working files, use the command *git rebase --abort* instead.

Assume the following history exists and the current branch is "topic":

```
          A---B---C topic
         /
    D---E---F---G master
```

From this point, the result of either of the following commands:

```
git rebase master
git rebase master topic
```

would be:

```
                  A'--B'--C' topic
                 /
    D---E---F---G master
```

NOTE: The latter form is just a short-hand of *git checkout topic* followed by *git rebase master*. When rebase exits *topic* will remain the checked-out branch.

If the upstream branch already contains a change you have made (e.g., because you mailed a patch which was applied upstream), then that commit will be skipped and warnings will be issued (if the *merge* backend is used). For example, running *git rebase master* on the following history (in which *A'* and *A* introduce the same set of changes, but have different committer information):

```
          A---B---C topic
         /
    D---E---A'---F master
```

will result in:

```
                 B'---C' topic
                /
D---E---A'---F master
```

Here is how you would transplant a topic branch based on one branch to another, to pretend that you forked the topic branch from the latter branch, using *rebase --onto*.

First let's assume your *topic* is based on branch *next*. For example, a feature developed in *topic* depends on some functionality which is found in *next*.

```
o---o---o---o---o  master
     \
      o---o---o---o---o  next
                       \
                        o---o---o  topic
```

We want to make *topic* forked from branch *master*; for example, because the functionality on which *topic* depends was merged into the more stable *master* branch. We want our tree to look like this:

```
o---o---o---o---o  master
    |            \
    |             o'--o'--o'  topic
     \
      o---o---o---o---o  next
```

We can get this using the following command:

```
git rebase --onto master next topic
```

Another example of --onto option is to rebase part of a branch. If we have the following situation:

```
                           H---I---J topicB
                          /
                 E---F---G  topicA
                /
A---B---C---D  master
```

then the command

```
git rebase --onto master topicA topicB
```

would result in:

```
                H'--I'--J'  topicB
               /
               | E---F---G  topicA
               |/
A---B---C---D  master
```

This is useful when topicB does not depend on topicA.

A range of commits could also be removed with rebase. If we have the following situation:

```
E---F---G---H---I---J  topicA
```

then the command

```
git rebase --onto topicA~5 topicA~3 topicA
```

would result in the removal of commits F and G:

```
    E---H'---I'---J'  topicA
```

This is useful if F and G were flawed in some way, or should not be part of topicA. Note that the argument to *--onto* and the <*upstream*> parameter can be any valid commit-ish.

In case of conflict, *git rebase* will stop at the first problematic commit and leave conflict markers in the tree. You can use *git diff* to locate the markers (<<<<<<) and make edits to resolve the conflict. For each file you edit, you need to tell Git that the conflict has been resolved, typically this would be done with

```
git add <filename>
```

After resolving the conflict manually and updating the index with the desired resolution, you can continue the rebasing process with

```
git rebase --continue
```

Alternatively, you can undo the *git rebase* with

```
git rebase --abort
```

MODE OPTIONS

The options in this section cannot be used with any other option, including not with each other:

--continue

Restart the rebasing process after having resolved a merge conflict.

--skip

Restart the rebasing process by skipping the current patch.

--abort

Abort the rebase operation and reset HEAD to the original branch. If <*branch*> was provided when the rebase operation was started, then *HEAD* will be reset to <*branch*>. Otherwise *HEAD* will be reset to where it was when the rebase operation was started.

--quit

Abort the rebase operation but *HEAD* is not reset back to the original branch. The index and working tree are also left unchanged as a result. If a temporary stash entry was created using *--autostash*, it will be saved to the stash list.

--edit-todo

Edit the todo list during an interactive rebase.

--show-current-patch

Show the current patch in an interactive rebase or when rebase is stopped because of conflicts. This is the equivalent of *git show REBASE_HEAD*.

OPTIONS

--onto <newbase>

Starting point at which to create the new commits. If the *--onto* option is not specified, the starting point is <*upstream*>. May be any valid commit, and not just an existing branch name.

As a special case, you may use "A...B" as a shortcut for the merge base of A and B if there is exactly one merge base. You can leave out at most one of A and B, in which case it defaults to HEAD.

--keep-base

Set the starting point at which to create the new commits to the merge base of <*upstream*> and <*branch*>. Running *git rebase --keep-base* <*upstream*> <*branch*> is equivalent to running *git rebase --reapply-cherry-picks --no-fork-point --onto* <*upstream*>...<*branch*> <*upstream*> <*branch*>.

This option is useful in the case where one is developing a feature on top of an upstream branch. While the feature is being worked on, the upstream branch may advance and it may not be the best idea to keep rebasing on top of the upstream but to keep the base commit as-is. As the base commit is unchanged this option implies *--reapply-cherry-picks* to avoid losing commits.

Although both this option and *--fork-point* find the merge base between <*upstream*> and <*branch*>, this option uses the merge base as the *starting point* on which new commits will be created, whereas *--fork-point* uses the merge base to determine the *set of commits* which will be rebased.

See also INCOMPATIBLE OPTIONS below.

<upstream>

Upstream branch to compare against. May be any valid commit, not just an existing branch name. Defaults to the configured upstream for the current branch.

<branch>

Working branch; defaults to *HEAD*.

--apply

Use applying strategies to rebase (calling *git-am* internally). This option may become a no-op in the future once the merge backend handles everything the apply one does.

See also INCOMPATIBLE OPTIONS below.

--empty=(drop|keep|stop)

How to handle commits that are not empty to start and are not clean cherry-picks of any upstream commit, but which become empty after rebasing (because they contain a subset of already upstream changes):

drop

The commit will be dropped. This is the default behavior.

keep

The commit will be kept. This option is implied when *--exec* is specified unless *-i*/*--interactive* is also specified.

stop , *ask*

The rebase will halt when the commit is applied, allowing you to choose whether to drop it, edit files more, or just commit the empty changes. This option is implied when *-i*/*--interactive* is specified. *ask* is a deprecated synonym of *stop*.

Note that commits which start empty are kept (unless *--no-keep-empty* is specified), and commits which are clean cherry-picks (as determined by *git log --cherry-mark* ...) are detected and dropped as a preliminary step (unless *--reapply-cherry-picks* or *--keep-base* is passed).

See also INCOMPATIBLE OPTIONS below.

--no-keep-empty , --keep-empty

Do not keep commits that start empty before the rebase (i.e. that do not change anything from its parent) in the result. The default is to keep commits which start empty, since creating such commits requires passing the

--allow-empty override flag to *git commit*, signifying that a user is very intentionally creating such a commit and thus wants to keep it.

Usage of this flag will probably be rare, since you can get rid of commits that start empty by just firing up an interactive rebase and removing the lines corresponding to the commits you don't want. This flag exists as a convenient shortcut, such as for cases where external tools generate many empty commits and you want them all removed.

For commits which do not start empty but become empty after rebasing, see the *--empty* flag.

See also INCOMPATIBLE OPTIONS below.

--reapply-cherry-picks , --no-reapply-cherry-picks

Reapply all clean cherry-picks of any upstream commit instead of preemptively dropping them. (If these commits then become empty after rebasing, because they contain a subset of already upstream changes, the behavior towards them is controlled by the *--empty* flag.)

In the absence of *--keep-base* (or if *--no-reapply-cherry-picks* is given), these commits will be automatically dropped. Because this necessitates reading all upstream commits, this can be expensive in repositories with a large number of upstream commits that need to be read. When using the *merge* backend, warnings will be issued for each dropped commit (unless *--quiet* is given). Advice will also be issued unless *advice.skippedCherryPicks* is set to false (see Section G.3.29, “git-config(1)”).

--reapply-cherry-picks allows rebase to forgo reading all upstream commits, potentially improving performance.

See also INCOMPATIBLE OPTIONS below.

--allow-empty-message

No-op. Rebasing commits with an empty message used to fail and this option would override that behavior, allowing commits with empty messages to be rebased. Now commits with an empty message do not cause rebasing to halt.

See also INCOMPATIBLE OPTIONS below.

-m , --merge

Using merging strategies to rebase (default).

Note that a rebase merge works by replaying each commit from the working branch on top of the <*upstream*> branch. Because of this, when a merge conflict happens, the side reported as *ours* is the so-far rebased series, starting with <*upstream*>, and *theirs* is the working branch. In other words, the sides are swapped.

See also INCOMPATIBLE OPTIONS below.

-s <strategy> , --strategy=<strategy>

Use the given merge strategy, instead of the default *ort*. This implies *--merge*.

Because *git rebase* replays each commit from the working branch on top of the <*upstream*> branch using the given strategy, using the *ours* strategy simply empties all patches from the <*branch*>, which makes little sense.

See also INCOMPATIBLE OPTIONS below.

-X <strategy-option> , --strategy-option=<strategy-option>

Pass the <strategy-option> through to the merge strategy. This implies *--merge* and, if no strategy has been specified, *-s ort*. Note the reversal of *ours* and *theirs* as noted above for the *-m* option.

See also INCOMPATIBLE OPTIONS below.

--rerere-autoupdate , --no-rerere-autoupdate

After the rerere mechanism reuses a recorded resolution on the current conflict to update the files in the working tree, allow it to also update the index with the result of resolution. *--no-rerere-autoupdate* is a good way to double-check what *rerere* did and catch potential mismerges, before committing the result to the index with a separate *git add*.

-S[<keyid>] , --gpg-sign[=<keyid>] , --no-gpg-sign

GPG-sign commits. The *keyid* argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space. *--no-gpg-sign* is useful to countermand both *commit.gpgSign* configuration variable, and earlier *--gpg-sign*.

-q , --quiet

Be quiet. Implies *--no-stat*.

-v , --verbose

Be verbose. Implies *--stat*.

--stat

Show a diffstat of what changed upstream since the last rebase. The diffstat is also controlled by the configuration option rebase.stat.

-n , --no-stat

Do not show a diffstat as part of the rebase process.

--no-verify

This option bypasses the pre-rebase hook. See also Section G.4.7, “githooks(5)”.

--verify

Allows the pre-rebase hook to run, which is the default. This option can be used to override *--no-verify*. See also Section G.4.7, “githooks(5)”.

-C<n>

Ensure at least <*n*> lines of surrounding context match before and after each change. When fewer lines of surrounding context exist they all must match. By default no context is ever ignored. Implies *--apply*.

See also INCOMPATIBLE OPTIONS below.

--no-ff , --force-rebase , -f

Individually replay all rebased commits instead of fast-forwarding over the unchanged ones. This ensures that the entire history of the rebased branch is composed of new commits.

You may find this helpful after reverting a topic branch merge, as this option recreates the topic branch with fresh commits so it can be remerged successfully without needing to "revert the reversion" (see the *revert-a-faulty-merge How-To* [https://www.kernel.org/pub/software/scm/git/docs/howto/revert-a-faulty-merge.html] for details).

--fork-point , --no-fork-point

Use reflog to find a better common ancestor between <*upstream*> and <*branch*> when calculating which commits have been introduced by <*branch*>.

When *--fork-point* is active, *fork_point* will be used instead of <*upstream*> to calculate the set of commits to rebase, where *fork_point* is the result of *git merge-base --fork-point* <*upstream*> <*branch*> command (see

Section G.3.81, “git-merge-base(1)”). If *fork_point* ends up being empty, the <*upstream*> will be used as a fallback.

If <*upstream*> or *--keep-base* is given on the command line, then the default is *--no-fork-point*, otherwise the default is *--fork-point*. See also *rebase.forkpoint* in Section G.3.29, “git-config(1)”.

If your branch was based on <*upstream*> but <*upstream*> was rewound and your branch contains commits which were dropped, this option can be used with *--keep-base* in order to drop those commits from your branch.

See also INCOMPATIBLE OPTIONS below.

--ignore-whitespace

Ignore whitespace differences when trying to reconcile differences. Currently, each backend implements an approximation of this behavior:

apply backend

When applying a patch, ignore changes in whitespace in context lines. Unfortunately, this means that if the "old" lines being replaced by the patch differ only in whitespace from the existing file, you will get a merge conflict instead of a successful patch application.

merge backend

Treat lines with only whitespace changes as unchanged when merging. Unfortunately, this means that any patch hunks that were intended to modify whitespace and nothing else will be dropped, even if the other side had no changes that conflicted.

--whitespace=<option>

This flag is passed to the *git apply* program (see Section G.3.5, “git-apply(1)”) that applies the patch. Implies *--apply*.

See also INCOMPATIBLE OPTIONS below.

--committer-date-is-author-date

Instead of using the current time as the committer date, use the author date of the commit being rebased as the committer date. This option implies *--force-rebase*.

--ignore-date , --reset-author-date

Instead of using the author date of the original commit, use the current time as the author date of the rebased commit. This option implies *--force-rebase*.

See also INCOMPATIBLE OPTIONS below.

--signoff

Add a *Signed-off-by* trailer to all the rebased commits. Note that if *--interactive* is given then only commits marked to be picked, edited or reworded will have the trailer added.

See also INCOMPATIBLE OPTIONS below.

-i , --interactive

Make a list of the commits which are about to be rebased. Let the user edit that list before rebasing. This mode can also be used to split commits (see SPLITTING COMMITS below).

The commit list format can be changed by setting the configuration option rebase.instructionFormat. A customized instruction format will automatically have the commit hash prepended to the format.

See also INCOMPATIBLE OPTIONS below.

-r , --rebase-merges[=(rebase-cousins|no-rebase-cousins)] , --no-rebase-merges

By default, a rebase will simply drop merge commits from the todo list, and put the rebased commits into a single, linear branch. With *--rebase-merges*, the rebase will instead try to preserve the branching structure within the commits that are to be rebased, by recreating the merge commits. Any resolved merge conflicts or manual amendments in these merge commits will have to be resolved/re-applied manually. *--no-rebase-merges* can be used to countermand both the *rebase.rebaseMerges* config option and a previous *--rebase-merges*.

When rebasing merges, there are two modes: *rebase-cousins* and *no-rebase-cousins*. If the mode is not specified, it defaults to *no-rebase-cousins*. In *no-rebase-cousins* mode, commits which do not have <*upstream*> as direct ancestor will keep their original branch point, i.e. commits that would be excluded by Section G.3.74, “git-log(1)”'s *--ancestry-path* option will keep their original ancestry by default. In *rebase-cousins* mode, such commits are instead rebased onto <*upstream*> (or <*onto*>, if specified).

It is currently only possible to recreate the merge commits using the *ort* merge strategy; different merge strategies can be used only via explicit *exec git merge -s <strategy> [...]* commands.

See also REBASING MERGES and INCOMPATIBLE OPTIONS below.

-x <cmd> , --exec <cmd>

Append "exec <cmd>" after each line creating a commit in the final history. <*cmd*> will be interpreted as one or more shell commands. Any command that fails will interrupt the rebase, with exit code 1.

You may execute several commands by either using one instance of *--exec* with several commands:

```
git rebase -i --exec "cmd1 && cmd2 && ..."
```

or by giving more than one *--exec*:

```
git rebase -i --exec "cmd1" --exec "cmd2" --exec ...
```

If *--autosquash* is used, *exec* lines will not be appended for the intermediate commits, and will only appear at the end of each squash/fixup series.

This uses the *--interactive* machinery internally, but it can be run without an explicit *--interactive*.

See also INCOMPATIBLE OPTIONS below.

--root

Rebase all commits reachable from <*branch*>, instead of limiting them with an <*upstream*>. This allows you to rebase the root commit(s) on a branch.

See also INCOMPATIBLE OPTIONS below.

--autosquash , --no-autosquash

Automatically squash commits with specially formatted messages into previous commits being rebased. If a commit message starts with "squash! ", "fixup! " or "amend! ", the remainder of the subject line is taken as a commit specifier, which matches a previous commit if it matches the subject line or the hash of that commit. If no commit matches fully, matches of the specifier with the start of commit subjects are considered.

In the rebase todo list, the actions of squash, fixup and amend commits are changed from *pick* to *squash*, *fixup* or *fixup -C*, respectively, and they are moved right after the commit they modify. The *--interactive* option can be used to review and edit the todo list before proceeding.

The recommended way to create commits with squash markers is by using the *--squash*, *--fixup*, *--fixup=amend:* or *--fixup=reword:* options of Section G.3.28, “git-commit(1)”, which take the target commit as an argument and automatically fill in the subject line of the new commit from that.

Setting configuration variable *rebase.autoSquash* to true enables auto-squashing by default for interactive rebase. The *--no-autosquash* option can be used to override that setting.

See also INCOMPATIBLE OPTIONS below.

--autostash , --no-autostash

Automatically create a temporary stash entry before the operation begins, and apply it after the operation ends. This means that you can run rebase on a dirty worktree. However, use with care: the final stash application after a successful rebase might result in non-trivial conflicts.

--reschedule-failed-exec , --no-reschedule-failed-exec

Automatically reschedule *exec* commands that failed. This only makes sense in interactive mode (or when an *--exec* option was provided).

This option applies once a rebase is started. It is preserved for the whole rebase based on, in order, the command line option provided to the initial *git rebase*, the *rebase.rescheduleFailedExec* configuration (see Section G.3.29, “git-config(1)” or "CONFIGURATION" below), or it defaults to false.

Recording this option for the whole rebase is a convenience feature. Otherwise an explicit *--no-reschedule-failed-exec* at the start would be overridden by the presence of a *rebase.rescheduleFailedExec=true* configuration when *git rebase --continue* is invoked. Currently, you cannot pass *--[no-]reschedule-failed-exec* to *git rebase --continue*.

--update-refs , --no-update-refs

Automatically force-update any branches that point to commits that are being rebased. Any branches that are checked out in a worktree are not updated in this way.

If the configuration variable *rebase.updateRefs* is set, then this option can be used to override and disable this setting.

See also INCOMPATIBLE OPTIONS below.

INCOMPATIBLE OPTIONS

The following options:

- --apply
- --whitespace
- -C

are incompatible with the following options:

- --merge
- --strategy
- --strategy-option
- --autosquash
- --rebase-merges
- --interactive
- --exec
- --no-keep-empty

- --empty=
- --[no-]reapply-cherry-picks when used without --keep-base
- --update-refs
- --root when used without --onto

In addition, the following pairs of options are incompatible:

- --keep-base and --onto
- --keep-base and --root
- --fork-point and --root

BEHAVIORAL DIFFERENCES

git rebase has two primary backends: *apply* and *merge*. (The *apply* backend used to be known as the *am* backend, but the name led to confusion as it looks like a verb instead of a noun. Also, the *merge* backend used to be known as the interactive backend, but it is now used for non-interactive cases as well. Both were renamed based on lower-level functionality that underpinned each.) There are some subtle differences in how these two backends behave:

1. Empty commits

The *apply* backend unfortunately drops intentionally empty commits, i.e. commits that started empty, though these are rare in practice. It also drops commits that become empty and has no option for controlling this behavior.

The *merge* backend keeps intentionally empty commits by default (though with *-i* they are marked as empty in the todo list editor, or they can be dropped automatically with *--no-keep-empty*).

Similar to the apply backend, by default the merge backend drops commits that become empty unless *-i*/*--interactive* is specified (in which case it stops and asks the user what to do). The merge backend also has an *--empty=(drop|keep|stop)* option for changing the behavior of handling commits that become empty.

2. Directory rename detection

Due to the lack of accurate tree information (arising from constructing fake ancestors with the limited information available in patches), directory rename detection is disabled in the *apply* backend. Disabled directory rename detection means that if one side of history renames a directory and the other adds new files to the old directory, then the new files will be left behind in the old directory without any warning at the time of rebasing that you may want to move these files into the new directory.

Directory rename detection works with the *merge* backend to provide you warnings in such cases.

3. Context

The *apply* backend works by creating a sequence of patches (by calling *format-patch* internally), and then applying the patches in sequence (calling *am* internally). Patches are composed of multiple hunks, each with line numbers, a context region, and the actual changes. The line numbers have to be taken with some fuzz, since the other side will likely have inserted or deleted lines earlier in the file. The context region is meant to help find how to adjust the line numbers in order to apply the changes to the right lines. However, if multiple areas of the code have the same surrounding lines of context, the wrong one can be picked. There are real-world cases where this has caused commits to be reapplied incorrectly with no conflicts reported. Setting *diff.context* to a larger value may prevent such types of problems, but increases the chance of spurious conflicts (since it will require more lines of matching context to apply).

The *merge* backend works with a full copy of each relevant file, insulating it from these types of problems.

4. Labelling of conflicts markers

When there are content conflicts, the merge machinery tries to annotate each side's conflict markers with the commits where the content came from. Since the *apply* backend drops the original information about the rebased commits and their parents (and instead generates new fake commits based off limited information in the generated patches), those commits cannot be identified; instead it has to fall back to a commit summary. Also, when *merge.conflictStyle* is set to *diff3* or *zdiff3*, the *apply* backend will use "constructed merge base" to label the content from the merge base, and thus provide no information about the merge base commit whatsoever.

The *merge* backend works with the full commits on both sides of history and thus has no such limitations.

5. Hooks

The *apply* backend has not traditionally called the post-commit hook, while the *merge* backend has. Both have called the post-checkout hook, though the *merge* backend has squelched its output. Further, both backends only call the post-checkout hook with the starting point commit of the rebase, not the intermediate commits nor the final commit. In each case, the calling of these hooks was by accident of implementation rather than by design (both backends were originally implemented as shell scripts and happened to invoke other commands like *git checkout* or *git commit* that would call the hooks). Both backends should have the same behavior, though it is not entirely clear which, if any, is correct. We will likely make rebase stop calling either of these hooks in the future.

6. Interruptability

The *apply* backend has safety problems with an ill-timed interrupt; if the user presses Ctrl-C at the wrong time to try to abort the rebase, the rebase can enter a state where it cannot be aborted with a subsequent *git rebase --abort*. The *merge* backend does not appear to suffer from the same shortcoming. (See https://lore.kernel.org/git/20200207132152.GC2868@szeder.dev/ for details.)

7. Commit Rewording

When a conflict occurs while rebasing, rebase stops and asks the user to resolve. Since the user may need to make notable changes while resolving conflicts, after conflicts are resolved and the user has run *git rebase --continue*, the rebase should open an editor and ask the user to update the commit message. The *merge* backend does this, while the *apply* backend blindly applies the original commit message.

8. Miscellaneous differences

There are a few more behavioral differences that most folks would probably consider inconsequential but which are mentioned for completeness:

- Reflog: The two backends will use different wording when describing the changes made in the reflog, though both will make use of the word "rebase".
- Progress, informational, and error messages: The two backends provide slightly different progress and informational messages. Also, the apply backend writes error messages (such as "Your files would be overwritten…") to stdout, while the merge backend writes them to stderr.
- State directories: The two backends keep their state in different directories under *.git/*

MERGE STRATEGIES

The merge mechanism (*git merge* and *git pull* commands) allows the backend *merge strategies* to be chosen with *-s* option. Some strategies can also take their own options, which can be passed by giving *-X<option>* arguments to *git merge* and/or *git pull*.

ort

This is the default merge strategy when pulling or merging one branch. This strategy can only resolve two heads using a 3-way merge algorithm. When there is more than one common ancestor that can be used for 3-

way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the 3-way merge. This has been reported to result in fewer merge conflicts without causing mismerges by tests done on actual merge commits taken from Linux 2.6 kernel development history. Additionally this strategy can detect and handle merges involving renames. It does not make use of detected copies. The name for this algorithm is an acronym ("Ostensibly Recursive's Twin") and came from the fact that it was written as a replacement for the previous default algorithm, *recursive*.

The *ort* strategy can take the following options:

ours

This option forces conflicting hunks to be auto-resolved cleanly by favoring *our* version. Changes from the other tree that do not conflict with our side are reflected in the merge result. For a binary file, the entire contents are taken from our side.

This should not be confused with the *ours* merge strategy, which does not even look at what the other tree contains at all. It discards everything the other tree did, declaring *our* history contains all that happened in it.

theirs

This is the opposite of *ours*; note that, unlike *ours*, there is no *theirs* merge strategy to confuse this merge option with.

ignore-space-change , ignore-all-space , ignore-space-at-eol , ignore-cr-at-eol

Treats lines with the indicated type of whitespace change as unchanged for the sake of a three-way merge. Whitespace changes mixed with other changes to a line are not ignored. See also Section G.3.44, “git-diff(1)” *-b*, *-w*, *--ignore-space-at-eol*, and *--ignore-cr-at-eol*.

- If *their* version only introduces whitespace changes to a line, *our* version is used;
- If *our* version introduces whitespace changes but *their* version includes a substantial change, *their* version is used;
- Otherwise, the merge proceeds in the usual way.

renormalize

This runs a virtual check-out and check-in of all three stages of a file when resolving a three-way merge. This option is meant to be used when merging branches with different clean filters or end-of-line normalization rules. See "Merging branches with differing checkin/checkout attributes" in Section G.4.2, “gitattributes(5)” for details.

no-renormalize

Disables the *renormalize* option. This overrides the *merge.renormalize* configuration variable.

find-renames[=<n>]

Turn on rename detection, optionally setting the similarity threshold. This is the default. This overrides the *merge.renames* configuration variable. See also Section G.3.44, “git-diff(1)” *--find-renames*.

rename-threshold=<n>

Deprecated synonym for *find-renames=<n>*.

subtree[=<path>]

This option is a more advanced form of *subtree* strategy, where the strategy makes a guess on how two trees must be shifted to match with each other when merging. Instead, the specified path is prefixed (or stripped from the beginning) to make the shape of two trees to match.

recursive

This can only resolve two heads using a 3-way merge algorithm. When there is more than one common ancestor that can be used for 3-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the 3-way merge. This has been reported to result in fewer merge conflicts without causing mismerges by tests done on actual merge commits taken from Linux 2.6 kernel development history. Additionally this can detect and handle merges involving renames. It does not make use of detected copies. This was the default strategy for resolving two heads from Git v0.99.9k until v2.33.0.

The *recursive* strategy takes the same options as *ort*. However, there are three additional options that *ort* ignores (not documented above) that are potentially useful with the *recursive* strategy:

patience

Deprecated synonym for *diff-algorithm=patience*.

diff-algorithm=[patience|minimal|histogram|myers]

Use a different diff algorithm while merging, which can help avoid mismerges that occur due to unimportant matching lines (such as braces from distinct functions). See also Section G.3.44, “git-diff(1)” *--diff-algorithm*. Note that *ort* specifically uses *diff-algorithm=histogram*, while *recursive* defaults to the *diff.algorithm* config setting.

no-renames

Turn off rename detection. This overrides the *merge.renames* configuration variable. See also Section G.3.44, “git-diff(1)” *--no-renames*.

resolve

This can only resolve two heads (i.e. the current branch and another branch you pulled from) using a 3-way merge algorithm. It tries to carefully detect criss-cross merge ambiguities. It does not handle renames.

octopus

This resolves cases with more than two heads, but refuses to do a complex merge that needs manual resolution. It is primarily meant to be used for bundling topic branch heads together. This is the default merge strategy when pulling or merging more than one branch.

ours

This resolves any number of heads, but the resulting tree of the merge is always that of the current branch head, effectively ignoring all changes from all other branches. It is meant to be used to supersede old development history of side branches. Note that this is different from the -Xours option to the *recursive* merge strategy.

subtree

This is a modified *ort* strategy. When merging trees A and B, if B corresponds to a subtree of A, B is first adjusted to match the tree structure of A, instead of reading the trees at the same level. This adjustment is also done to the common ancestor tree.

With the strategies that use 3-way merge (including the default, *ort*), if a change is made on both branches, but later reverted on one of the branches, that change will be present in the merged result; some people find this behavior confusing. It occurs because only the heads and the merge base are considered when performing a merge, not the individual commits. The merge algorithm therefore considers the reverted change as no change at all, and substitutes the changed version instead.

NOTES

You should understand the implications of using *git rebase* on a repository that you share. See also RECOVERING FROM UPSTREAM REBASE below.

When the rebase is run, it will first execute a *pre-rebase* hook if one exists. You can use this hook to do sanity checks and reject the rebase if it isn't appropriate. Please see the template *pre-rebase* hook script for an example.

Upon completion, <*branch*> will be the current branch.

INTERACTIVE MODE

Rebasing interactively means that you have a chance to edit the commits which are rebased. You can reorder the commits, and you can remove them (weeding out bad or otherwise unwanted patches).

The interactive mode is meant for this type of workflow:

1. have a wonderful idea
2. hack on the code
3. prepare a series for submission
4. submit

where point 2. consists of several instances of

a) regular use

1. finish something worthy of a commit
2. commit

b) independent fixup

1. realize that something does not work
2. fix that
3. commit it

Sometimes the thing fixed in b.2. cannot be amended to the not-quite perfect commit it fixes, because that commit is buried deeply in a patch series. That is exactly what interactive rebase is for: use it after plenty of "a"s and "b"s, by rearranging and editing commits, and squashing multiple commits into one.

Start it with the last commit you want to retain as-is:

```
git rebase -i <after-this-commit>
```

An editor will be fired up with all the commits in your current branch (ignoring merge commits), which come after the given commit. You can reorder the commits in this list to your heart's content, and you can remove them. The list looks more or less like this:

```
pick deadbee The oneline of this commit
pick fa1afe1 The oneline of the next commit
...
```

The oneline descriptions are purely for your pleasure; *git rebase* will not look at them but at the commit names ("deadbee" and "fa1afe1" in this example), so do not delete or edit the names.

By replacing the command "pick" with the command "edit", you can tell *git rebase* to stop after applying that commit, so that you can edit the files and/or the commit message, amend the commit, and continue rebasing.

To interrupt the rebase (just like an "edit" command would do, but without cherry-picking any commit first), use the "break" command.

If you just want to edit the commit message for a commit, replace the command "pick" with the command "reword".

To drop a commit, replace the command "pick" with "drop", or just delete the matching line.

If you want to fold two or more commits into one, replace the command "pick" for the second and subsequent commits with "squash" or "fixup". If the commits had different authors, the folded commit will be attributed to the author of the first commit. The suggested commit message for the folded commit is the concatenation of the first commit's message with those identified by "squash" commands, omitting the messages of commits identified by "fixup" commands, unless "fixup -c" is used. In that case the suggested commit message is only the message of the "fixup -c" commit, and an editor is opened allowing you to edit the message. The contents (patch) of the "fixup -c" commit are still incorporated into the folded commit. If there is more than one "fixup -c" commit, the message from the final one is used. You can also use "fixup -C" to get the same behavior as "fixup -c" except without opening an editor.

git rebase will stop when "pick" has been replaced with "edit" or when a command fails due to merge errors. When you are done editing and/or resolving conflicts you can continue with *git rebase --continue*.

For example, if you want to reorder the last 5 commits, such that what was *HEAD~4* becomes the new *HEAD*. To achieve that, you would call *git rebase* like this:

```
$ git rebase -i HEAD~5
```

And move the first patch to the end of the list.

You might want to recreate merge commits, e.g. if you have a history like this:

```
           X
            \
         A---M---B
        /
---o---O---P---Q
```

Suppose you want to rebase the side branch starting at "A" to "Q". Make sure that the current *HEAD* is "B", and call

```
$ git rebase -i -r --onto Q O
```

Reordering and editing commits usually creates untested intermediate steps. You may want to check that your history editing did not break anything by running a test, or at least recompiling at intermediate points in history by using the "exec" command (shortcut "x"). You may do so by creating a todo list like this one:

```
pick deadbee Implement feature XXX
fixup f1a5c00 Fix to feature XXX
exec make
pick c0ffeee The oneline of the next commit
edit deadbab The oneline of the commit after
exec cd subdir; make test
...
```

The interactive rebase will stop when a command fails (i.e. exits with non-0 status) to give you an opportunity to fix the problem. You can continue with *git rebase --continue*.

The "exec" command launches the command in a shell (the default one, usually /bin/sh), so you can use shell features (like "cd", ">", ";" …). The command is run from the root of the working tree.

```
$ git rebase -i --exec "make test"
```

This command lets you check that intermediate commits are compilable. The todo list becomes like that:

```
pick 5928aea one
exec make test
pick 04d0fda two
exec make test
```

```
pick ba46169 three
exec make test
pick f4593f9 four
exec make test
```

SPLITTING COMMITS

In interactive mode, you can mark commits with the action "edit". However, this does not necessarily mean that *git rebase* expects the result of this edit to be exactly one commit. Indeed, you can undo the commit, or you can add other commits. This can be used to split a commit into two:

- Start an interactive rebase with *git rebase -i <commit>^*, where *<commit>* is the commit you want to split. In fact, any commit range will do, as long as it contains that commit.
- Mark the commit you want to split with the action "edit".
- When it comes to editing that commit, execute *git reset HEAD^*. The effect is that the *HEAD* is rewound by one, and the index follows suit. However, the working tree stays the same.
- Now add the changes to the index that you want to have in the first commit. You can use *git add* (possibly interactively) or *git gui* (or both) to do that.
- Commit the now-current index with whatever commit message is appropriate now.
- Repeat the last two steps until your working tree is clean.
- Continue the rebase with *git rebase --continue*.

If you are not absolutely sure that the intermediate revisions are consistent (they compile, pass the testsuite, etc.) you should use *git stash* to stash away the not-yet-committed changes after each commit, test, and amend the commit if fixes are necessary.

RECOVERING FROM UPSTREAM REBASE

Rebasing (or any other form of rewriting) a branch that others have based work on is a bad idea: anyone downstream of it is forced to manually fix their history. This section explains how to do the fix from the downstream's point of view. The real fix, however, would be to avoid rebasing the upstream in the first place.

To illustrate, suppose you are in a situation where someone develops a *subsystem* branch, and you are working on a *topic* that is dependent on this *subsystem*. You might end up with a history like the following:

```
o---o---o---o---o---o---o---o  master
     \
      o---o---o---o---o  subsystem
                       \
                        *---*---*  topic
```

If *subsystem* is rebased against *master*, the following happens:

```
o---o---o---o---o---o---o---o  master
     \                       \
      o---o---o---o---o       o'--o'--o'--o'--o'  subsystem
                       \
                        *---*---*  topic
```

If you now continue development as usual, and eventually merge *topic* to *subsystem*, the commits from *subsystem* will remain duplicated forever:

```
o---o---o---o---o---o---o---o  master
     \                       \
```

```
o---o---o---o---o       o'--o'--o'--o'--o'--M  subsystem
                 \                           /
                  *---*---*-..........-*--*  topic
```

Such duplicates are generally frowned upon because they clutter up history, making it harder to follow. To clean things up, you need to transplant the commits on *topic* to the new *subsystem* tip, i.e., rebase *topic*. This becomes a ripple effect: anyone downstream from *topic* is forced to rebase too, and so on!

There are two kinds of fixes, discussed in the following subsections:

Easy case: The changes are literally the same.

This happens if the *subsystem* rebase was a simple rebase and had no conflicts.

Hard case: The changes are not the same.

This happens if the *subsystem* rebase had conflicts, or used *--interactive* to omit, edit, squash, or fixup commits; or if the upstream used one of *commit --amend*, *reset*, or a full history rewriting command like *filter-repo* [https://github.com/newren/git-filter-repo].

1. The easy case

Only works if the changes (patch IDs based on the diff contents) on *subsystem* are literally the same before and after the rebase *subsystem* did.

In that case, the fix is easy because *git rebase* knows to skip changes that are already present in the new upstream (unless *--reapply-cherry-picks* is given). So if you say (assuming you're on *topic*)

```
$ git rebase subsystem
```

you will end up with the fixed history

```
o---o---o---o---o---o---o---o  master
                            \
                             o'--o'--o'--o'  subsystem
                                          \
                                           *---*---*  topic
```

2. The hard case

Things get more complicated if the *subsystem* changes do not exactly correspond to the ones before the rebase.

> **Note**
>
> While an "easy case recovery" sometimes appears to be successful even in the hard case, it may have unintended consequences. For example, a commit that was removed via *git rebase --interactive* will be **resurrected**!

The idea is to manually tell *git rebase* "where the old *subsystem* ended and your *topic* began", that is, what the old merge base between them was. You will have to find a way to name the last commit of the old *subsystem*, for example:

- With the *subsystem* reflog: after *git fetch*, the old tip of *subsystem* is at *subsystem@{1}*. Subsequent fetches will increase the number. (See Section G.3.109, “git-reflog(1)”.)
- Relative to the tip of *topic*: knowing that your *topic* has three commits, the old tip of *subsystem* must be *topic~3*.

You can then transplant the old *subsystem..topic* to the new tip by saying (for the reflog case, and assuming you are on *topic* already):

```
$ git rebase --onto subsystem subsystem@{1}
```

The ripple effect of a "hard case" recovery is especially bad: *everyone* downstream from *topic* will now have to perform a "hard case" recovery too!

REBASING MERGES

The interactive rebase command was originally designed to handle individual patch series. As such, it makes sense to exclude merge commits from the todo list, as the developer may have merged the then-current *master* while working on the branch, only to rebase all the commits onto *master* eventually (skipping the merge commits).

However, there are legitimate reasons why a developer may want to recreate merge commits: to keep the branch structure (or "commit topology") when working on multiple, inter-related branches.

In the following example, the developer works on a topic branch that refactors the way buttons are defined, and on another topic branch that uses that refactoring to implement a "Report a bug" button. The output of *git log --graph --format=%s -5* may look like this:

```
*   Merge branch 'report-a-bug'
|\
| * Add the feedback button
* | Merge branch 'refactor-button'
|\ \
| |/
| * Use the Button class for all buttons
| * Extract a generic Button class from the DownloadButton one
```

The developer might want to rebase those commits to a newer *master* while keeping the branch topology, for example when the first topic branch is expected to be integrated into *master* much earlier than the second one, say, to resolve merge conflicts with changes to the DownloadButton class that made it into *master*.

This rebase can be performed using the *--rebase-merges* option. It will generate a todo list looking like this:

```
label onto

# Branch: refactor-button
reset onto
pick 123456 Extract a generic Button class from the DownloadButton one
pick 654321 Use the Button class for all buttons
label refactor-button

# Branch: report-a-bug
reset refactor-button # Use the Button class for all buttons
pick abcdef Add the feedback button
label report-a-bug

reset onto
merge -C a1b2c3 refactor-button # Merge 'refactor-button'
merge -C 6f5e4d report-a-bug # Merge 'report-a-bug'
```

In contrast to a regular interactive rebase, there are *label*, *reset* and *merge* commands in addition to *pick* ones.

The *label* command associates a label with the current HEAD when that command is executed. These labels are created as worktree-local refs (*refs/rewritten/<label>*) that will be deleted when the rebase finishes. That way, rebase operations in multiple worktrees linked to the same repository do not interfere with one another. If the *label* command fails, it is rescheduled immediately, with a helpful message how to proceed.

The *reset* command resets the HEAD, index and worktree to the specified revision. It is similar to an *exec git reset --hard <label>*, but refuses to overwrite untracked files. If the *reset* command fails, it is rescheduled immediately,

with a helpful message how to edit the todo list (this typically happens when a *reset* command was inserted into the todo list manually and contains a typo).

The *merge* command will merge the specified revision(s) into whatever is HEAD at that time. With *-C <original-commit>*, the commit message of the specified merge commit will be used. When the *-C* is changed to a lower-case *-c*, the message will be opened in an editor after a successful merge so that the user can edit the message.

If a *merge* command fails for any reason other than merge conflicts (i.e. when the merge operation did not even start), it is rescheduled immediately.

By default, the *merge* command will use the *ort* merge strategy for regular merges, and *octopus* for octopus merges. One can specify a default strategy for all merges using the *--strategy* argument when invoking rebase, or can override specific merges in the interactive list of commands by using an *exec* command to call *git merge* explicitly with a *--strategy* argument. Note that when calling *git merge* explicitly like this, you can make use of the fact that the labels are worktree-local refs (the ref *refs/rewritten/onto* would correspond to the label *onto*, for example) in order to refer to the branches you want to merge.

Note: the first command (*label onto*) labels the revision onto which the commits are rebased; The name *onto* is just a convention, as a nod to the *--onto* option.

It is also possible to introduce completely new merge commits from scratch by adding a command of the form *merge <merge-head>*. This form will generate a tentative commit message and always open an editor to let the user edit it. This can be useful e.g. when a topic branch turns out to address more than a single concern and wants to be split into two or even more topic branches. Consider this todo list:

```
pick 192837 Switch from GNU Makefiles to CMake
pick 5a6c7e Document the switch to CMake
pick 918273 Fix detection of OpenSSL in CMake
pick afbecd http: add support for TLS v1.3
pick fdbaec Fix detection of cURL in CMake on Windows
```

The one commit in this list that is not related to CMake may very well have been motivated by working on fixing all those bugs introduced by switching to CMake, but it addresses a different concern. To split this branch into two topic branches, the todo list could be edited like this:

```
label onto

pick afbecd http: add support for TLS v1.3
label tlsv1.3

reset onto
pick 192837 Switch from GNU Makefiles to CMake
pick 918273 Fix detection of OpenSSL in CMake
pick fdbaec Fix detection of cURL in CMake on Windows
pick 5a6c7e Document the switch to CMake
label cmake

reset onto
merge tlsv1.3
merge cmake
```

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

rebase.backend

Default backend to use for rebasing. Possible choices are *apply* or *merge*. In the future, if the merge backend gains all remaining capabilities of the apply backend, this setting may become unused.

rebase.stat

Whether to show a diffstat of what changed upstream since the last rebase. False by default.

rebase.autoSquash

If set to true, enable the *--autosquash* option of Section G.3.107, “git-rebase(1)” by default for interactive mode. This can be overridden with the *--no-autosquash* option.

rebase.autoStash

When set to true, automatically create a temporary stash entry before the operation begins, and apply it after the operation ends. This means that you can run rebase on a dirty worktree. However, use with care: the final stash application after a successful rebase might result in non-trivial conflicts. This option can be overridden by the *--no-autostash* and *--autostash* options of Section G.3.107, “git-rebase(1)”. Defaults to false.

rebase.updateRefs

If set to true enable *--update-refs* option by default.

rebase.missingCommitsCheck

If set to "warn", git rebase -i will print a warning if some commits are removed (e.g. a line was deleted), however the rebase will still proceed. If set to "error", it will print the previous warning and stop the rebase, *git rebase --edit-todo* can then be used to correct the error. If set to "ignore", no checking is done. To drop a commit without warning or error, use the *drop* command in the todo list. Defaults to "ignore".

rebase.instructionFormat

A format string, as specified in Section G.3.74, “git-log(1)”, to be used for the todo list during an interactive rebase. The format will automatically have the commit hash prepended to the format.

rebase.abbreviateCommands

If set to true, *git rebase* will use abbreviated command names in the todo list resulting in something like this:

```
p deadbee The oneline of the commit
p fa1afe1 The oneline of the next commit
...
```

instead of:

```
pick deadbee The oneline of the commit
pick fa1afe1 The oneline of the next commit
...
```

Defaults to false.

rebase.rescheduleFailedExec

Automatically reschedule *exec* commands that failed. This only makes sense in interactive mode (or when an *--exec* option was provided). This is the same as specifying the *--reschedule-failed-exec* option.

rebase.forkPoint

If set to false set *--no-fork-point* option by default.

rebase.rebaseMerges

Whether and how to set the *--rebase-merges* option by default. Can be *rebase-cousins*, *no-rebase-cousins*, or a boolean. Setting to true or to *no-rebase-cousins* is equivalent to *--rebase-merges=no-rebase-cousins*, setting to *rebase-cousins* is equivalent to *--rebase-merges=rebase-cousins*, and setting to false is equivalent to *--no-*

rebase-merges. Passing *--rebase-merges* on the command line, with or without an argument, overrides any *rebase.rebaseMerges* configuration.

rebase.maxLabelLength

When generating label names from commit subjects, truncate the names to this length. By default, the names are truncated to a little less than *NAME_MAX* (to allow e.g. *.lock* files to be written for the corresponding loose refs).

sequence.editor

Text editor used by *git rebase -i* for editing the rebase instruction file. The value is meant to be interpreted by the shell when it is used. It can be overridden by the *GIT_SEQUENCE_EDITOR* environment variable. When not configured, the default commit message editor is used instead.

GIT

Part of the Section G.3.1, "git(1)" suite

G.3.108. git-receive-pack(1)

2024-07-02

Revision History

2024-07-02

NAME

git-receive-pack - Receive what is pushed into the repository

SYNOPSIS

git receive-pack <git-dir>

DESCRIPTION

Invoked by *git send-pack* and updates the repository with the information fed from the remote end.

This command is usually not invoked directly by the end user. The UI for the protocol is on the *git send-pack* side, and the program pair is meant to be used to push updates to a remote repository. For pull operations, see Section G.3.48, "git-fetch-pack(1)".

The command allows for the creation and fast-forwarding of sha1 refs (heads/tags) on the remote end (strictly speaking, it is the local end *git-receive-pack* runs, but to the user who is sitting at the send-pack end, it is updating the remote. Confused?)

There are other real-world examples of using update and post-update hooks found in the Documentation/howto directory.

git-receive-pack honours the receive.denyNonFastForwards config option, which tells it if updates to a ref should be denied if they are not fast-forwards.

A number of other receive.* config options are available to tweak its behavior, see Section G.3.29, "git-config(1)".

OPTIONS

<git-dir>

The repository to sync into.

--http-backend-info-refs

Used by Section G.3.65, "git-http-backend(1)" to serve up *$GIT_URL/info/refs?service=git-receive-pack* requests. See *--http-backend-info-refs* in Section G.3.151, "git-upload-pack(1)".

PRE-RECEIVE HOOK

Before any ref is updated, if $GIT_DIR/hooks/pre-receive file exists and is executable, it will be invoked once with no parameters. The standard input of the hook will be one line per ref to be updated:

```
sha1-old SP sha1-new SP refname LF
```

The refname value is relative to $GIT_DIR; e.g. for the master head this is "refs/heads/master". The two sha1 values before each refname are the object names for the refname before and after the update. Refs to be created will have sha1-old equal to 0{40}, while refs to be deleted will have sha1-new equal to 0{40}, otherwise sha1-old and sha1-new should be valid objects in the repository.

When accepting a signed push (see Section G.3.103, “git-push(1)”), the signed push certificate is stored in a blob and an environment variable *GIT_PUSH_CERT* can be consulted for its object name. See the description of *post-receive* hook for an example. In addition, the certificate is verified using GPG and the result is exported with the following environment variables:

GIT_PUSH_CERT_SIGNER

The name and the e-mail address of the owner of the key that signed the push certificate.

GIT_PUSH_CERT_KEY

The GPG key ID of the key that signed the push certificate.

GIT_PUSH_CERT_STATUS

The status of GPG verification of the push certificate, using the same mnemonic as used in *%G?* format of *git log* family of commands (see Section G.3.74, “git-log(1)”).

GIT_PUSH_CERT_NONCE

The nonce string the process asked the signer to include in the push certificate. If this does not match the value recorded on the "nonce" header in the push certificate, it may indicate that the certificate is a valid one that is being replayed from a separate "git push" session.

GIT_PUSH_CERT_NONCE_STATUS

UNSOLICITED

"git push --signed" sent a nonce when we did not ask it to send one.

MISSING

"git push --signed" did not send any nonce header.

BAD

"git push --signed" sent a bogus nonce.

OK

"git push --signed" sent the nonce we asked it to send.

SLOP

"git push --signed" sent a nonce different from what we asked it to send now, but in a previous session. See *GIT_PUSH_CERT_NONCE_SLOP* environment variable.

GIT_PUSH_CERT_NONCE_SLOP

"git push --signed" sent a nonce different from what we asked it to send now, but in a different session whose starting time is different by this many seconds from the current session. Only meaningful when

GIT_PUSH_CERT_NONCE_STATUS says *SLOP*. Also read about *receive.certNonceSlop* variable in Section G.3.29, “git-config(1)”.

This hook is called before any refname is updated and before any fast-forward checks are performed.

If the pre-receive hook exits with a non-zero exit status no updates will be performed, and the update, post-receive and post-update hooks will not be invoked either. This can be useful to quickly bail out if the update is not to be supported.

See the notes on the quarantine environment below.

UPDATE HOOK

Before each ref is updated, if $GIT_DIR/hooks/update file exists and is executable, it is invoked once per ref, with three parameters:

```
$GIT_DIR/hooks/update refname sha1-old sha1-new
```

The refname parameter is relative to $GIT_DIR; e.g. for the master head this is "refs/heads/master". The two sha1 arguments are the object names for the refname before and after the update. Note that the hook is called before the refname is updated, so either sha1-old is 0{40} (meaning there is no such ref yet), or it should match what is recorded in refname.

The hook should exit with non-zero status if it wants to disallow updating the named ref. Otherwise it should exit with zero.

Successful execution (a zero exit status) of this hook does not ensure the ref will actually be updated, it is only a prerequisite. As such it is not a good idea to send notices (e.g. email) from this hook. Consider using the post-receive hook instead.

POST-RECEIVE HOOK

After all refs were updated (or attempted to be updated), if any ref update was successful, and if $GIT_DIR/hooks/post-receive file exists and is executable, it will be invoked once with no parameters. The standard input of the hook will be one line for each successfully updated ref:

```
sha1-old SP sha1-new SP refname LF
```

The refname value is relative to $GIT_DIR; e.g. for the master head this is "refs/heads/master". The two sha1 values before each refname are the object names for the refname before and after the update. Refs that were created will have sha1-old equal to 0{40}, while refs that were deleted will have sha1-new equal to 0{40}, otherwise sha1-old and sha1-new should be valid objects in the repository.

The *GIT_PUSH_CERT** environment variables can be inspected, just as in *pre-receive* hook, after accepting a signed push.

Using this hook, it is easy to generate mails describing the updates to the repository. This example script sends one mail message per ref listing the commits pushed to the repository, and logs the push certificates of signed pushes with good signatures to a logger service:

```
#!/bin/sh
# mail out commit update information.
while read oval nval ref
do
        if expr "$oval" : '0*$' >/dev/null
        then
                echo "Created a new ref, with the following commits:"
                git rev-list --pretty "$nval"
        else
                echo "New commits:"
                git rev-list --pretty "$nval" "^$oval"
```

```
        fi |
        mail -s "Changes to ref $ref" commit-list@mydomain
done
# log signed push certificate, if any
if test -n "${GIT_PUSH_CERT-}" && test ${GIT_PUSH_CERT_STATUS} = G
then
        (
                echo expected nonce is ${GIT_PUSH_NONCE}
                git cat-file blob ${GIT_PUSH_CERT}
        ) | mail -s "push certificate from $GIT_PUSH_CERT_SIGNER" push-
log@mydomain
fi
exit 0
```

The exit code from this hook invocation is ignored, however a non-zero exit code will generate an error message.

Note that it is possible for refname to not have sha1-new when this hook runs. This can easily occur if another user modifies the ref after it was updated by *git-receive-pack*, but before the hook was able to evaluate it. It is recommended that hooks rely on sha1-new rather than the current value of refname.

POST-UPDATE HOOK

After all other processing, if at least one ref was updated, and if $GIT_DIR/hooks/post-update file exists and is executable, then post-update will be called with the list of refs that have been updated. This can be used to implement any repository wide cleanup tasks.

The exit code from this hook invocation is ignored; the only thing left for *git-receive-pack* to do at that point is to exit itself anyway.

This hook can be used, for example, to run *git update-server-info* if the repository is packed and is served via a dumb transport.

```
#!/bin/sh
exec git update-server-info
```

QUARANTINE ENVIRONMENT

When *receive-pack* takes in objects, they are placed into a temporary "quarantine" directory within the *$GIT_DIR/objects* directory and migrated into the main object store only after the *pre-receive* hook has completed. If the push fails before then, the temporary directory is removed entirely.

This has a few user-visible effects and caveats:

1. Pushes which fail due to problems with the incoming pack, missing objects, or due to the *pre-receive* hook will not leave any on-disk data. This is usually helpful to prevent repeated failed pushes from filling up your disk, but can make debugging more challenging.

2. Any objects created by the *pre-receive* hook will be created in the quarantine directory (and migrated only if it succeeds).

3. The *pre-receive* hook MUST NOT update any refs to point to quarantined objects. Other programs accessing the repository will not be able to see the objects (and if the pre-receive hook fails, those refs would become corrupted). For safety, any ref updates from within *pre-receive* are automatically rejected.

SEE ALSO

Section G.3.125, “git-send-pack(1)”, Section G.4.11, “gitnamespaces(7)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.109. git-reflog(1)

2024-07-02

Revision History

2024-07-02

NAME

git-reflog - Manage reflog information

SYNOPSIS

git reflog [show] [<log-options>] [<ref>]
git reflog list
git reflog expire [--expire=<time>] [--expire-unreachable=<time>]
[--rewrite] [--updateref] [--stale-fix]
[--dry-run | -n] [--verbose] [--all [--single-worktree] | <refs>…]
git reflog delete [--rewrite] [--updateref]
[--dry-run | -n] [--verbose] <ref>@{<specifier>}…
git reflog exists <ref>

DESCRIPTION

This command manages the information recorded in the reflogs.

Reference logs, or "reflogs", record when the tips of branches and other references were updated in the local repository. Reflogs are useful in various Git commands, to specify the old value of a reference. For example, *HEAD@{2}* means "where HEAD used to be two moves ago", *master@{one.week.ago}* means "where master used to point to one week ago in this local repository", and so on. See Section G.4.14, “gitrevisions(7)” for more details.

The command takes various subcommands, and different options depending on the subcommand:

The "show" subcommand (which is also the default, in the absence of any subcommands) shows the log of the reference provided in the command-line (or *HEAD*, by default). The reflog covers all recent actions, and in addition the *HEAD* reflog records branch switching. *git reflog show* is an alias for *git log -g --abbrev-commit --pretty=oneline*; see Section G.3.74, “git-log(1)” for more information.

The "list" subcommand lists all refs which have a corresponding reflog.

The "expire" subcommand prunes older reflog entries. Entries older than *expire* time, or entries older than *expire-unreachable* time and not reachable from the current tip, are removed from the reflog. This is typically not used directly by end users -- instead, see Section G.3.58, “git-gc(1)”.

The "delete" subcommand deletes single entries from the reflog. Its argument must be an *exact* entry (e.g. "*git reflog delete master@{2}*"). This subcommand is also typically not used directly by end users.

The "exists" subcommand checks whether a ref has a reflog. It exits with zero status if the reflog exists, and non-zero status if it does not.

OPTIONS

1. Options for *show*

git reflog show accepts any of the options accepted by *git log*.

2. Options for *expire*

--all

Process the reflogs of all references.

--single-worktree

By default when --*all* is specified, reflogs from all working trees are processed. This option limits the processing to reflogs from the current working tree only.

--expire=<time>

Prune entries older than the specified time. If this option is not specified, the expiration time is taken from the configuration setting *gc.reflogExpire*, which in turn defaults to 90 days. --*expire=all* prunes entries regardless of their age; --*expire=never* turns off pruning of reachable entries (but see --*expire-unreachable*).

--expire-unreachable=<time>

Prune entries older than <*time*> that are not reachable from the current tip of the branch. If this option is not specified, the expiration time is taken from the configuration setting *gc.reflogExpireUnreachable*, which in turn defaults to 30 days. --*expire-unreachable=all* prunes unreachable entries regardless of their age; --*expire-unreachable=never* turns off early pruning of unreachable entries (but see --*expire*).

--updateref

Update the reference to the value of the top reflog entry (i.e. <ref>@{0}) if the previous top entry was pruned. (This option is ignored for symbolic references.)

--rewrite

If a reflog entry's predecessor is pruned, adjust its "old" SHA-1 to be equal to the "new" SHA-1 field of the entry that now precedes it.

--stale-fix

Prune any reflog entries that point to "broken commits". A broken commit is a commit that is not reachable from any of the reference tips and that refers, directly or indirectly, to a missing commit, tree, or blob object.

This computation involves traversing all the reachable objects, i.e. it has the same cost as *git prune*. It is primarily intended to fix corruption caused by garbage collecting using older versions of Git, which didn't protect objects referred to by reflogs.

-n , --dry-run

Do not actually prune any entries; just show what would have been pruned.

--verbose

Print extra information on screen.

3. Options for *delete*

git reflog delete accepts options --*updateref*, --*rewrite*, -*n*, --*dry-run*, and --*verbose*, with the same meanings as when they are used with *expire*.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.110. git-remote-ext(1)

2024-07-02
Revision History

2024-07-02

NAME

git-remote-ext - Bridge smart transport to external command.

SYNOPSIS

git remote add <nick> "ext::<command>[<arguments>…]"

DESCRIPTION

This remote helper uses the specified *<command>* to connect to a remote Git server.

Data written to stdin of the specified *<command>* is assumed to be sent to a git:// server, git-upload-pack, git-receive-pack or git-upload-archive (depending on situation), and data read from stdout of <command> is assumed to be received from the same service.

Command and arguments are separated by an unescaped space.

The following sequences have a special meaning:

'% '

Literal space in command or argument.

%%

Literal percent sign.

%s

Replaced with name (receive-pack, upload-pack, or upload-archive) of the service Git wants to invoke.

%S

Replaced with long name (git-receive-pack, git-upload-pack, or git-upload-archive) of the service Git wants to invoke.

%G (must be the first characters in an argument)

This argument will not be passed to *<command>*. Instead, it will cause the helper to start by sending git:// service requests to the remote side with the service field set to an appropriate value and the repository field set to the rest of the argument. Default is not to send such a request.

This is useful if the remote side is git:// server accessed over some tunnel.

%V (must be first characters in argument)

This argument will not be passed to *<command>*. Instead it sets the vhost field in the git:// service request (to the rest of the argument). Default is not to send vhost in such request (if sent).

ENVIRONMENT VARIABLES

GIT_TRANSLOOP_DEBUG

If set, prints debugging information about various reads/writes.

ENVIRONMENT VARIABLES PASSED TO COMMAND

GIT_EXT_SERVICE

Set to long name (git-upload-pack, etc…) of service helper needs to invoke.

GIT_EXT_SERVICE_NOPREFIX

Set to long name (upload-pack, etc…) of service helper needs to invoke.

EXAMPLES

This remote helper is transparently used by Git when you use commands such as "git fetch <URL>", "git clone <URL>", , "git push <URL>" or "git remote add <nick> <URL>", where <URL> begins with *ext::*. Examples:

"ext::ssh -i /home/foo/.ssh/somekey user@host.example %S *foo/repo*"

Like host.example:foo/repo, but use /home/foo/.ssh/somekey as keypair and user as the user on the remote side. This avoids the need to edit .ssh/config.

"ext::socat -t3600 - ABSTRACT-CONNECT:/git-server %G/somerepo"

Represents repository with path /somerepo accessible over git protocol at the abstract namespace address /git-server.

"ext::git-server-alias foo %G/repo"

Represents a repository with path /repo accessed using the helper program "git-server-alias foo". The path to the repository and type of request are not passed on the command line but as part of the protocol stream, as usual with git:// protocol.

"ext::git-server-alias foo %G/repo %Vfoo"

Represents a repository with path /repo accessed using the helper program "git-server-alias foo". The hostname for the remote server passed in the protocol stream will be "foo" (this allows multiple virtual Git servers to share a link-level address).

"ext::git-server-alias foo %G/repo% with% spaces %Vfoo"

Represents a repository with path */repo with spaces* accessed using the helper program "git-server-alias foo". The hostname for the remote server passed in the protocol stream will be "foo" (this allows multiple virtual Git servers to share a link-level address).

"ext::git-ssl foo.example /bar"

Represents a repository accessed using the helper program "git-ssl foo.example /bar". The type of request can be determined by the helper using environment variables (see above).

SEE ALSO

Section G.4.12, “gitremote-helpers(7)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.111. git-remote-fd(1)

2024-07-02

Revision History

2024-07-02

NAME

git-remote-fd - Reflect smart transport stream back to caller

SYNOPSIS

"fd::<infd>[,<outfd>][/<anything>]" (as URL)

DESCRIPTION

This helper uses specified file descriptors to connect to a remote Git server. This is not meant for end users but for programs and scripts calling git fetch, push, or archive.

If only <infd> is given, it is assumed to be a bidirectional socket connected to a remote Git server (git-upload-pack, git-receive-pack, or git-upload-archive). If both <infd> and <outfd> are given, they are assumed to be pipes connected to a remote Git server (<infd> being the inbound pipe and <outfd> being the outbound pipe).

It is assumed that any handshaking procedures have already been completed (such as sending service request for git://) before this helper is started.

<anything> can be any string. It is ignored. It is meant for providing information to the user in the URL in case that URL is displayed in some context.

ENVIRONMENT VARIABLES

GIT_TRANSLOOP_DEBUG

If set, prints debugging information about various reads/writes.

EXAMPLES

git fetch fd::17 master

Fetch master, using file descriptor #17 to communicate with git-upload-pack.

git fetch fd::17/foo master

Same as above.

git push fd::7,8 master (as URL)

Push master, using file descriptor #7 to read data from git-receive-pack and file descriptor #8 to write data to the same service.

git push fd::7,8/bar master

Same as above.

SEE ALSO

Section G.4.12, “gitremote-helpers(7)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.112. git-remote(1)

2024-07-02
Revision History
2024-07-02

NAME

git-remote - Manage set of tracked repositories

SYNOPSIS

git remote [-v | --verbose]
git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=(fetch|push)] <name> <URL>
git remote rename [--[no-]progress] <old> <new>
git remote remove <name>
git remote set-head <name> (-a | --auto | -d | --delete | <branch>)
git remote set-branches [--add] <name> <branch>…
git remote get-url [--push] [--all] <name>

git remote set-url [--push] <name> <newurl> [<oldurl>]
git remote set-url --add [--push] <name> <newurl>
git remote set-url --delete [--push] <name> <URL>
git remote [-v | --verbose] *show* [-n] <name>…
git remote prune [-n | --dry-run] <name>…
git remote [-v | --verbose] *update* [-p | --prune] [(<group> | <remote>)…]

DESCRIPTION

Manage the set of repositories ("remotes") whose branches you track.

OPTIONS

-v , --verbose

Be a little more verbose and show remote url after name. For promisor remotes, also show which filters (*blob:none* etc.) are configured. NOTE: This must be placed between *remote* and subcommand.

COMMANDS

With no arguments, shows a list of existing remotes. Several subcommands are available to perform operations on the remotes.

add

Add a remote named <name> for the repository at <URL>. The command *git fetch <name>* can then be used to create and update remote-tracking branches <name>/<branch>.

With *-f* option, *git fetch <name>* is run immediately after the remote information is set up.

With *--tags* option, *git fetch <name>* imports every tag from the remote repository.

With *--no-tags* option, *git fetch <name>* does not import tags from the remote repository.

By default, only tags on fetched branches are imported (see Section G.3.49, “git-fetch(1)”).

With *-t <branch>* option, instead of the default glob refspec for the remote to track all branches under the *refs/remotes/<name>/* namespace, a refspec to track only *<branch>* is created. You can give more than one *-t <branch>* to track multiple branches without grabbing all branches.

With *-m <master>* option, a symbolic-ref *refs/remotes/<name>/HEAD* is set up to point at remote's *<master>* branch. See also the set-head command.

When a fetch mirror is created with *--mirror=fetch*, the refs will not be stored in the *refs/remotes/* namespace, but rather everything in *refs/* on the remote will be directly mirrored into *refs/* in the local repository. This option only makes sense in bare repositories, because a fetch would overwrite any local commits.

When a push mirror is created with *--mirror=push*, then *git push* will always behave as if *--mirror* was passed.

rename

Rename the remote named <old> to <new>. All remote-tracking branches and configuration settings for the remote are updated.

In case <old> and <new> are the same, and <old> is a file under *$GIT_DIR/remotes* or *$GIT_DIR/branches*, the remote is converted to the configuration file format.

remove , *rm*

Remove the remote named <name>. All remote-tracking branches and configuration settings for the remote are removed.

set-head

Sets or deletes the default branch (i.e. the target of the symbolic-ref *refs/remotes/<name>/HEAD*) for the named remote. Having a default branch for a remote is not required, but allows the name of the remote to be specified in lieu of a specific branch. For example, if the default branch for *origin* is set to *master*, then *origin* may be specified wherever you would normally specify *origin/master*.

With *-d* or *--delete*, the symbolic ref *refs/remotes/<name>/HEAD* is deleted.

With *-a* or *--auto*, the remote is queried to determine its *HEAD*, then the symbolic-ref *refs/remotes/<name>/HEAD* is set to the same branch. e.g., if the remote *HEAD* is pointed at *next*, *git remote set-head origin -a* will set the symbolic-ref *refs/remotes/origin/HEAD* to *refs/remotes/origin/next*. This will only work if *refs/remotes/origin/next* already exists; if not it must be fetched first.

Use *<branch>* to set the symbolic-ref *refs/remotes/<name>/HEAD* explicitly. e.g., *git remote set-head origin master* will set the symbolic-ref *refs/remotes/origin/HEAD* to *refs/remotes/origin/master*. This will only work if *refs/remotes/origin/master* already exists; if not it must be fetched first.

set-branches

Changes the list of branches tracked by the named remote. This can be used to track a subset of the available remote branches after the initial setup for a remote.

The named branches will be interpreted as if specified with the *-t* option on the *git remote add* command line.

With *--add*, instead of replacing the list of currently tracked branches, adds to that list.

get-url

Retrieves the URLs for a remote. Configurations for *insteadOf* and *pushInsteadOf* are expanded here. By default, only the first URL is listed.

With *--push*, push URLs are queried rather than fetch URLs.

With *--all*, all URLs for the remote will be listed.

set-url

Changes URLs for the remote. Sets first URL for remote <name> that matches regex <oldurl> (first URL if no <oldurl> is given) to <newurl>. If <oldurl> doesn't match any URL, an error occurs and nothing is changed.

With *--push*, push URLs are manipulated instead of fetch URLs.

With *--add*, instead of changing existing URLs, new URL is added.

With *--delete*, instead of changing existing URLs, all URLs matching regex <URL> are deleted for remote <name>. Trying to delete all non-push URLs is an error.

Note that the push URL and the fetch URL, even though they can be set differently, must still refer to the same place. What you pushed to the push URL should be what you would see if you immediately fetched from the fetch URL. If you are trying to fetch from one place (e.g. your upstream) and push to another (e.g. your publishing repository), use two separate remotes.

show

Gives some information about the remote <name>.

With *-n* option, the remote heads are not queried first with *git ls-remote <name>*; cached information is used instead.

prune

Deletes stale references associated with <name>. By default, stale remote-tracking branches under <name> are deleted, but depending on global configuration and the configuration of the remote we might even prune local

tags that haven't been pushed there. Equivalent to *git fetch --prune <name>*, except that no new references will be fetched.

See the PRUNING section of Section G.3.49, “git-fetch(1)” for what it'll prune depending on various configuration.

With *--dry-run* option, report what branches would be pruned, but do not actually prune them.

update

Fetch updates for remotes or remote groups in the repository as defined by *remotes.<group>*. If neither group nor remote is specified on the command line, the configuration parameter remotes.default will be used; if remotes.default is not defined, all remotes which do not have the configuration parameter *remote.<name>.skipDefaultUpdate* set to true will be updated. (See Section G.3.29, “git-config(1)”).

With *--prune* option, run pruning against all the remotes that are updated.

DISCUSSION

The remote configuration is achieved using the *remote.origin.url* and *remote.origin.fetch* configuration variables. (See Section G.3.29, “git-config(1)”).

EXIT STATUS

On success, the exit status is *0*.

When subcommands such as *add*, *rename*, and *remove* can't find the remote in question, the exit status is *2*. When the remote already exists, the exit status is *3*.

On any other error, the exit status may be any other non-zero value.

EXAMPLES

- Add a new remote, fetch, and check out a branch from it

```
$ git remote
origin
$ git branch -r
  origin/HEAD -> origin/master
  origin/master
$ git remote add staging git://git.kernel.org/.../gregkh/staging.git
$ git remote
origin
staging
$ git fetch staging
...
From git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/staging
 * [new branch]      master     -> staging/master
 * [new branch]      staging-linus -> staging/staging-linus
 * [new branch]      staging-next -> staging/staging-next
$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  staging/master
  staging/staging-linus
  staging/staging-next
$ git switch -c staging staging/master
...
```

- Imitate *git clone* but track only selected branches

```
$ mkdir project.git
$ cd project.git
$ git init
$ git remote add -f -t master -m master origin git://example.com/git.git/
$ git merge origin
```

SEE ALSO

Section G.3.49, “git-fetch(1)” Section G.3.10, “git-branch(1)” Section G.3.29, “git-config(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.113. git-repack(1)

2024-07-02

Revision History

2024-07-02

NAME

git-repack - Pack unpacked objects in a repository

SYNOPSIS

git repack [-a] [-A] [-d] [-f] [-F] [-l] [-n] [-q] [-b] [-m] [--window=<n>] [--depth=<n>] [--threads=<n>] [--keep-pack=<pa

DESCRIPTION

This command is used to combine all objects that do not currently reside in a "pack", into a pack. It can also be used to re-organize existing packs into a single, more efficient pack.

A pack is a collection of objects, individually compressed, with delta compression applied, stored in a single file, with an associated index file.

Packs are used to reduce the load on mirror systems, backup engines, disk storage, etc.

OPTIONS

-a

Instead of incrementally packing the unpacked objects, pack everything referenced into a single pack. Especially useful when packing a repository that is used for private development. Use with *-d*. This will clean up the objects that *git prune* leaves behind, but *git fsck --full --dangling* shows as dangling.

Note that users fetching over dumb protocols will have to fetch the whole new pack in order to get any contained object, no matter how many other objects in that pack they already have locally.

Promisor packfiles are repacked separately: if there are packfiles that have an associated ".promisor" file, these packfiles will be repacked into another separate pack, and an empty ".promisor" file corresponding to the new separate pack will be written.

-A

Same as *-a*, unless *-d* is used. Then any unreachable objects in a previous pack become loose, unpacked objects, instead of being left in the old pack. Unreachable objects are never intentionally added to a pack, even when repacking. This option prevents unreachable objects from being immediately deleted by way of being left in the old pack and then removed. Instead, the loose unreachable objects will be pruned according to normal expiry rules with the next *git gc* invocation. See Section G.3.58, “git-gc(1)”.

-d

After packing, if the newly created packs make some existing packs redundant, remove the redundant packs. Also run *git prune-packed* to remove redundant loose object files.

--cruft

Same as *-a*, unless *-d* is used. Then any unreachable objects are packed into a separate cruft pack. Unreachable objects can be pruned using the normal expiry rules with the next *git gc* invocation (see Section G.3.58, "git-gc(1)"). Incompatible with *-k*.

--cruft-expiration=<approxidate>

Expire unreachable objects older than <*approxidate*> immediately instead of waiting for the next *git gc* invocation. Only useful with *--cruft -d*.

--max-cruft-size=<n>

Repack cruft objects into packs as large as <*n*> bytes before creating new packs. As long as there are enough cruft packs smaller than <*n*>, repacking will cause a new cruft pack to be created containing objects from any combined cruft packs, along with any new unreachable objects. Cruft packs larger than <*n*> will not be modified. When the new cruft pack is larger than <*n*> bytes, it will be split into multiple packs, all of which are guaranteed to be at most <*n*> bytes in size. Only useful with *--cruft -d*.

--expire-to=<dir>

Write a cruft pack containing pruned objects (if any) to the directory <*dir*>. This option is useful for keeping a copy of any pruned objects in a separate directory as a backup. Only useful with *--cruft -d*.

-l

Pass the *--local* option to *git pack-objects*. See Section G.3.96, "git-pack-objects(1)".

-f

Pass the *--no-reuse-delta* option to *git-pack-objects*, see Section G.3.96, "git-pack-objects(1)".

-F

Pass the *--no-reuse-object* option to *git-pack-objects*, see Section G.3.96, "git-pack-objects(1)".

-q , --quiet

Show no progress over the standard error stream and pass the *-q* option to *git pack-objects*. See Section G.3.96, "git-pack-objects(1)".

-n

Do not update the server information with *git update-server-info*. This option skips updating local catalog files needed to publish this repository (or a direct copy of it) over HTTP or FTP. See Section G.3.149, "git-update-server-info(1)".

--window=<n> , --depth=<n>

These two options affect how the objects contained in the pack are stored using delta compression. The objects are first internally sorted by type, size and optionally names and compared against the other objects within *--window* to see if using delta compression saves space. *--depth* limits the maximum delta depth; making it too deep affects the performance on the unpacker side, because delta data needs to be applied that many times to get to the necessary object.

The default value for --window is 10 and --depth is 50. The maximum depth is 4095.

--threads=<n>

This option is passed through to *git pack-objects*.

--window-memory=<n>

This option provides an additional limit on top of *--window*; the window size will dynamically scale down so as to not take up more than *<n>* bytes in memory. This is useful in repositories with a mix of large and small objects to not run out of memory with a large window, but still be able to take advantage of the large window for the smaller objects. The size can be suffixed with "k", "m", or "g". *--window-memory=0* makes memory usage unlimited. The default is taken from the *pack.windowMemory* configuration variable. Note that the actual memory usage will be the limit multiplied by the number of threads used by Section G.3.96, "git-pack-objects(1)".

--max-pack-size=<n>

Maximum size of each output pack file. The size can be suffixed with "k", "m", or "g". The minimum size allowed is limited to 1 MiB. If specified, multiple packfiles may be created, which also prevents the creation of a bitmap index. The default is unlimited, unless the config variable *pack.packSizeLimit* is set. Note that this option may result in a larger and slower repository; see the discussion in *pack.packSizeLimit*.

--filter=<filter-spec>

Remove objects matching the filter specification from the resulting packfile and put them into a separate packfile. Note that objects used in the working directory are not filtered out. So for the split to fully work, it's best to perform it in a bare repo and to use the *-a* and *-d* options along with this option. Also *--no-write-bitmap-index* (or the *repack.writebitmaps* config option set to *false*) should be used otherwise writing bitmap index will fail, as it supposes a single packfile containing all the objects. See Section G.3.120, "git-rev-list(1)" for valid *<filter-spec>* forms.

--filter-to=<dir>

Write the pack containing filtered out objects to the directory *<dir>*. Only useful with *--filter*. This can be used for putting the pack on a separate object directory that is accessed through the Git alternates mechanism. **WARNING:** If the packfile containing the filtered out objects is not accessible, the repo can become corrupt as it might not be possible to access the objects in that packfile. See the *objects* and *objects/info/alternates* sections of Section G.4.13, "gitrepository-layout(5)".

-b , --write-bitmap-index

Write a reachability bitmap index as part of the repack. This only makes sense when used with *-a*, *-A* or *-m*, as the bitmaps must be able to refer to all reachable objects. This option overrides the setting of *repack.writeBitmaps*. This option has no effect if multiple packfiles are created, unless writing a MIDX (in which case a multi-pack bitmap is created).

--pack-kept-objects

Include objects in *.keep* files when repacking. Note that we still do not delete *.keep* packs after *pack-objects* finishes. This means that we may duplicate objects, but this makes the option safe to use when there are concurrent pushes or fetches. This option is generally only useful if you are writing bitmaps with *-b* or *repack.writeBitmaps*, as it ensures that the bitmapped packfile has the necessary objects.

--keep-pack=<pack-name>

Exclude the given pack from repacking. This is the equivalent of having *.keep* file on the pack. *<pack-name>* is the pack file name without leading directory (e.g. *pack-123.pack*). The option can be specified multiple times to keep multiple packs.

--unpack-unreachable=<when>

When loosening unreachable objects, do not bother loosening any objects older than *<when>*. This can be used to optimize out the write of any objects that would be immediately pruned by a follow-up *git prune*.

-k , --keep-unreachable

When used with *-ad*, any unreachable objects from existing packs will be appended to the end of the packfile instead of being removed. In addition, any unreachable loose objects will be packed (and their loose counterparts removed).

-i , --delta-islands

Pass the *--delta-islands* option to *git-pack-objects*, see Section G.3.96, “git-pack-objects(1)”.

-g<factor> , --geometric=<factor>

Arrange resulting pack structure so that each successive pack contains at least <*factor*> times the number of objects as the next-largest pack.

git repack ensures this by determining a "cut" of packfiles that need to be repacked into one in order to ensure a geometric progression. It picks the smallest set of packfiles such that as many of the larger packfiles (by count of objects contained in that pack) may be left intact.

Unlike other repack modes, the set of objects to pack is determined uniquely by the set of packs being "rolled-up"; in other words, the packs determined to need to be combined in order to restore a geometric progression.

Loose objects are implicitly included in this "roll-up", without respect to their reachability. This is subject to change in the future.

When writing a multi-pack bitmap, *git repack* selects the largest resulting pack as the preferred pack for object selection by the MIDX (see Section G.3.92, “git-multi-pack-index(1)”).

-m , --write-midx

Write a multi-pack index (see Section G.3.92, “git-multi-pack-index(1)”) containing the non-redundant packs.

CONFIGURATION

Various configuration variables affect packing, see Section G.3.29, “git-config(1)” (search for "pack" and "delta").

By default, the command passes *--delta-base-offset* option to *git pack-objects*; this typically results in slightly smaller packs, but the generated packs are incompatible with versions of Git older than version 1.4.4. If you need to share your repository with such ancient Git versions, either directly or via the dumb http protocol, then you need to set the configuration variable *repack.UseDeltaBaseOffset* to "false" and repack. Access from old Git versions over the native protocol is unaffected by this option as the conversion is performed on the fly as needed in that case.

Delta compression is not used on objects larger than the *core.bigFileThreshold* configuration variable and on files with the attribute *delta* set to false.

SEE ALSO

Section G.3.96, “git-pack-objects(1)” Section G.3.100, “git-prune-packed(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.114. git-replace(1)

2024-07-02

Revision History

2024-07-02

NAME

git-replace - Create, list, delete refs to replace objects

SYNOPSIS

git replace [-f] <object> <replacement>
git replace [-f] --edit <object>
git replace [-f] --graft <commit> [<parent>…]
git replace [-f] --convert-graft-file
git replace -d <object>…
git replace [--format=<format>] [-l [<pattern>]]

DESCRIPTION

Adds a *replace* reference in *refs/replace/* namespace.

The name of the *replace* reference is the SHA-1 of the object that is replaced. The content of the *replace* reference is the SHA-1 of the replacement object.

The replaced object and the replacement object must be of the same type. This restriction can be bypassed using *-f*.

Unless *-f* is given, the *replace* reference must not yet exist.

There is no other restriction on the replaced and replacement objects. Merge commits can be replaced by non-merge commits and vice versa.

Replacement references will be used by default by all Git commands except those doing reachability traversal (prune, pack transfer and fsck).

It is possible to disable the use of replacement references for any command using the *--no-replace-objects* option just after *git*.

For example if commit *foo* has been replaced by commit *bar*:

```
$ git --no-replace-objects cat-file commit foo
```

shows information about commit *foo*, while:

```
$ git cat-file commit foo
```

shows information about commit *bar*.

The *GIT_NO_REPLACE_OBJECTS* environment variable can be set to achieve the same effect as the *--no-replace-objects* option.

OPTIONS

-f , --force

If an existing replace ref for the same object exists, it will be overwritten (instead of failing).

-d , --delete

Delete existing replace refs for the given objects.

--edit <object>

Edit an object's content interactively. The existing content for <object> is pretty-printed into a temporary file, an editor is launched on the file, and the result is parsed to create a new object of the same type as <object>. A replacement ref is then created to replace <object> with the newly created object. See Section G.3.152, “git-var(1)” for details about how the editor will be chosen.

--raw

When editing, provide the raw object contents rather than pretty-printed ones. Currently this only affects trees, which will be shown in their binary form. This is harder to work with, but can help when repairing a tree

that is so corrupted it cannot be pretty-printed. Note that you may need to configure your editor to cleanly read and write binary data.

--graft <commit> [<parent>…]

Create a graft commit. A new commit is created with the same content as <commit> except that its parents will be [<parent>…] instead of <commit>'s parents. A replacement ref is then created to replace <commit> with the newly created commit. Use *--convert-graft-file* to convert a *$GIT_DIR/info/grafts* file and use replace refs instead.

--convert-graft-file

Creates graft commits for all entries in *$GIT_DIR/info/grafts* and deletes that file upon success. The purpose is to help users with transitioning off of the now-deprecated graft file.

-l <pattern> , --list <pattern>

List replace refs for objects that match the given pattern (or all if no pattern is given). Typing "git replace" without arguments, also lists all replace refs.

--format=<format>

When listing, use the specified <format>, which can be one of *short*, *medium* and *long*. When omitted, the format defaults to *short*.

FORMATS

The following formats are available:

- *short*: <replaced-sha1>
- *medium*: <replaced-sha1> → <replacement-sha1>
- *long*: <replaced-sha1> (<replaced-type>) → <replacement-sha1> (<replacement-type>)

CREATING REPLACEMENT OBJECTS

Section G.3.62, “git-hash-object(1)”, Section G.3.107, “git-rebase(1)”, and git-filter-repo [https://github.com/newren/git-filter-repo], among other git commands, can be used to create replacement objects from existing objects. The *--edit* option can also be used with *git replace* to create a replacement object by editing an existing object.

If you want to replace many blobs, trees or commits that are part of a string of commits, you may just want to create a replacement string of commits and then only replace the commit at the tip of the target string of commits with the commit at the tip of the replacement string of commits.

BUGS

Comparing blobs or trees that have been replaced with those that replace them will not work properly. And using *git reset --hard* to go back to a replaced commit will move the branch to the replacement commit instead of the replaced commit.

There may be other problems when using *git rev-list* related to pending objects.

SEE ALSO

Section G.3.62, “git-hash-object(1)” Section G.3.107, “git-rebase(1)” Section G.3.144, “git-tag(1)” Section G.3.10, “git-branch(1)” Section G.3.28, “git-commit(1)” Section G.3.152, “git-var(1)” Section G.3.1, “git(1)” git-filter-repo [https://github.com/newren/git-filter-repo]

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.115. git-replay(1)

2024-07-02

Revision History

2024-07-02

NAME

git-replay - EXPERIMENTAL: Replay commits on a new base, works with bare repos too

SYNOPSIS

(EXPERIMENTAL!) *git replay* ([--contained] --onto <newbase> | --advance <branch>) <revision-range>…

DESCRIPTION

Takes ranges of commits and replays them onto a new location. Leaves the working tree and the index untouched, and updates no references. The output of this command is meant to be used as input to *git update-ref --stdin*, which would update the relevant branches (see the OUTPUT section below).

THIS COMMAND IS EXPERIMENTAL. THE BEHAVIOR MAY CHANGE.

OPTIONS

--onto <newbase>

Starting point at which to create the new commits. May be any valid commit, and not just an existing branch name.

When *--onto* is specified, the update-ref command(s) in the output will update the branch(es) in the revision range to point at the new commits, similar to the way how *git rebase --update-refs* updates multiple branches in the affected range.

--advance <branch>

Starting point at which to create the new commits; must be a branch name.

When *--advance* is specified, the update-ref command(s) in the output will update the branch passed as an argument to *--advance* to point at the new commits (in other words, this mimics a cherry-pick operation).

<revision-range>

Range of commits to replay. More than one <revision-range> can be passed, but in *--advance <branch>* mode, they should have a single tip, so that it's clear where <branch> should point to. See "Specifying Ranges" in Section G.3.121, “git-rev-parse(1)” and the "Commit Limiting" options below.

1. Commit Limiting

Besides specifying a range of commits that should be listed using the special notations explained in the description, additional commit limiting may be applied.

Using more options generally further limits the output (e.g. *--since=<date1>* limits to commits newer than *<date1>*, and using it with *--grep=<pattern>* further limits to commits whose log message has a line that matches *<pattern>*), unless otherwise noted.

Note that these are applied before commit ordering and formatting options, such as *--reverse*.

-<number> , -n <number> , --max-count=<number>

Limit the number of commits to output.

--skip=<number>

Skip *number* commits before starting to show the commit output.

--since=<date> , --after=<date>

Show commits more recent than a specific date.

--since-as-filter=<date>

Show all commits more recent than a specific date. This visits all commits in the range, rather than stopping at the first commit which is older than a specific date.

--until=<date> , --before=<date>

Show commits older than a specific date.

--author=<pattern> , --committer=<pattern>

Limit the commits output to ones with author/committer header lines that match the specified pattern (regular expression). With more than one *--author=<pattern>*, commits whose author matches any of the given patterns are chosen (similarly for multiple *--committer=<pattern>*).

--grep-reflog=<pattern>

Limit the commits output to ones with reflog entries that match the specified pattern (regular expression). With more than one *--grep-reflog*, commits whose reflog message matches any of the given patterns are chosen. It is an error to use this option unless *--walk-reflogs* is in use.

--grep=<pattern>

Limit the commits output to ones with a log message that matches the specified pattern (regular expression). With more than one *--grep=<pattern>*, commits whose message matches any of the given patterns are chosen (but see *--all-match*).

When *--notes* is in effect, the message from the notes is matched as if it were part of the log message.

--all-match

Limit the commits output to ones that match all given *--grep*, instead of ones that match at least one.

--invert-grep

Limit the commits output to ones with a log message that do not match the pattern specified with *--grep=<pattern>*.

-i , --regexp-ignore-case

Match the regular expression limiting patterns without regard to letter case.

--basic-regexp

Consider the limiting patterns to be basic regular expressions; this is the default.

-E , --extended-regexp

Consider the limiting patterns to be extended regular expressions instead of the default basic regular expressions.

-F , --fixed-strings

Consider the limiting patterns to be fixed strings (don't interpret pattern as a regular expression).

-P , --perl-regexp

Consider the limiting patterns to be Perl-compatible regular expressions.

Support for these types of regular expressions is an optional compile-time dependency. If Git wasn't compiled with support for them providing this option will cause it to die.

--remove-empty

Stop when a given path disappears from the tree.

--merges

Print only merge commits. This is exactly the same as *--min-parents=2*.

--no-merges

Do not print commits with more than one parent. This is exactly the same as *--max-parents=1*.

--min-parents=<number> , --max-parents=<number> , --no-min-parents , --no-max-parents

Show only commits which have at least (or at most) that many parent commits. In particular, *--max-parents=1* is the same as *--no-merges*, *--min-parents=2* is the same as *--merges*. *--max-parents=0* gives all root commits and *--min-parents=3* all octopus merges.

--no-min-parents and *--no-max-parents* reset these limits (to no limit) again. Equivalent forms are *--min-parents=0* (any commit has 0 or more parents) and *--max-parents=-1* (negative numbers denote no upper limit).

--first-parent

When finding commits to include, follow only the first parent commit upon seeing a merge commit. This option can give a better overview when viewing the evolution of a particular topic branch, because merges into a topic branch tend to be only about adjusting to updated upstream from time to time, and this option allows you to ignore the individual commits brought in to your history by such a merge.

--exclude-first-parent-only

When finding commits to exclude (with a ^), follow only the first parent commit upon seeing a merge commit. This can be used to find the set of changes in a topic branch from the point where it diverged from the remote branch, given that arbitrary merges can be valid topic branch changes.

--not

Reverses the meaning of the ^ prefix (or lack thereof) for all following revision specifiers, up to the next *--not*. When used on the command line before --stdin, the revisions passed through stdin will not be affected by it. Conversely, when passed via standard input, the revisions passed on the command line will not be affected by it.

--all

Pretend as if all the refs in *refs/*, along with *HEAD*, are listed on the command line as *<commit>*.

--branches[=<pattern>]

Pretend as if all the refs in *refs/heads* are listed on the command line as *<commit>*. If *<pattern>* is given, limit branches to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--tags[=<pattern>]

Pretend as if all the refs in *refs/tags* are listed on the command line as *<commit>*. If *<pattern>* is given, limit tags to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--remotes[=<pattern>]

Pretend as if all the refs in *refs/remotes* are listed on the command line as *<commit>*. If *<pattern>* is given, limit remote-tracking branches to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--glob=<glob-pattern>

Pretend as if all the refs matching shell glob <*glob-pattern*> are listed on the command line as <*commit*>. Leading *refs/*, is automatically prepended if missing. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--exclude=<glob-pattern>

Do not include refs matching <*glob-pattern*> that the next *--all*, *--branches*, *--tags*, *--remotes*, or *--glob* would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next *--all*, *--branches*, *--tags*, *--remotes*, or *--glob* option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with *refs/heads*, *refs/tags*, or *refs/remotes* when applied to *--branches*, *--tags*, or *--remotes*, respectively, and they must begin with *refs/* when applied to *--glob* or *--all*. If a trailing */** is intended, it must be given explicitly.

--exclude-hidden=[fetch|receive|uploadpack]

Do not include refs that would be hidden by *git-fetch*, *git-receive-pack* or *git-upload-pack* by consulting the appropriate *fetch.hideRefs*, *receive.hideRefs* or *uploadpack.hideRefs* configuration along with *transfer.hideRefs* (see Section G.3.29, “git-config(1)”). This option affects the next pseudo-ref option *--all* or *--glob* and is cleared after processing them.

--reflog

Pretend as if all objects mentioned by reflogs are listed on the command line as <*commit*>.

--alternate-refs

Pretend as if all objects mentioned as ref tips of alternate repositories were listed on the command line. An alternate repository is any repository whose object directory is specified in *objects/info/alternates*. The set of included objects may be modified by *core.alternateRefsCommand*, etc. See Section G.3.29, “git-config(1)”.

--single-worktree

By default, all working trees will be examined by the following options when there are more than one (see Section G.3.159, “git-worktree(1)”): *--all*, *--reflog* and *--indexed-objects*. This option forces them to examine the current working tree only.

--ignore-missing

Upon seeing an invalid object name in the input, pretend as if the bad input was not given.

--bisect

Pretend as if the bad bisection ref *refs/bisect/bad* was listed and as if it was followed by *--not* and the good bisection refs *refs/bisect/good-** on the command line.

--stdin

In addition to getting arguments from the command line, read them from standard input as well. This accepts commits and pseudo-options like *--all* and *--glob=*. When a -- separator is seen, the following input is treated as paths and used to limit the result. Flags like *--not* which are read via standard input are only respected for arguments passed in the same way and will not influence any subsequent command line arguments.

--cherry-mark

Like *--cherry-pick* (see below) but mark equivalent commits with = rather than omitting them, and inequivalent ones with +.

--cherry-pick

Omit any commit that introduces the same change as another commit on the other side when the set of commits are limited with symmetric difference.

For example, if you have two branches, *A* and *B*, a usual way to list all commits on only one side of them is with *--left-right* (see the example below in the description of the *--left-right* option). However, it shows the commits that were cherry-picked from the other branch (for example, 3rd on b may be cherry-picked from branch A). With this option, such pairs of commits are excluded from the output.

--left-only , --right-only

List only commits on the respective side of a symmetric difference, i.e. only those which would be marked < resp. > by *--left-right*.

For example, *--cherry-pick --right-only A...B* omits those commits from *B* which are in *A* or are patch-equivalent to a commit in *A*. In other words, this lists the + commits from *git cherry A B*. More precisely, *--cherry-pick --right-only --no-merges* gives the exact list.

--cherry

A synonym for *--right-only --cherry-mark --no-merges*; useful to limit the output to the commits on our side and mark those that have been applied to the other side of a forked history with *git log --cherry upstream...mybranch*, similar to *git cherry upstream mybranch*.

-g , --walk-reflogs

Instead of walking the commit ancestry chain, walk reflog entries from the most recent one to older ones. When this option is used you cannot specify commits to exclude (that is, *^commit*, *commit1..commit2*, and *commit1...commit2* notations cannot be used).

With *--pretty* format other than *oneline* and *reference* (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. The reflog designator in the output may be shown as *ref@{<Nth>}* (where *<Nth>* is the reverse-chronological index in the reflog) or as *ref@{<timestamp>}* (with the *<timestamp>* for that entry), depending on a few rules:

1. If the starting point is specified as *ref@{<Nth>}*, show the index format.
2. If the starting point was specified as *ref@{now}*, show the timestamp format.
3. If neither was used, but *--date* was given on the command line, show the timestamp in the format requested by *--date*.
4. Otherwise, show the index format.

Under *--pretty=oneline*, the commit message is prefixed with this information on the same line. This option cannot be combined with *--reverse*. See also Section G.3.109, “git-reflog(1)”.

Under *--pretty=reference*, this information will not be shown at all.

--merge

Show commits touching conflicted paths in the range *HEAD...<other>*, where *<other>* is the first existing pseudoref in *MERGE_HEAD*, *CHERRY_PICK_HEAD*, *REVERT_HEAD* or *REBASE_HEAD*. Only works when the index has unmerged entries. This option can be used to show relevant commits when resolving conflicts from a 3-way merge.

--boundary

Output excluded boundary commits. Boundary commits are prefixed with -.

2. History Simplification

Sometimes you are only interested in parts of the history, for example the commits modifying a particular <path>. But there are two parts of *History Simplification*, one part is selecting the commits and the other is how to do it, as there are various strategies to simplify the history.

The following options select the commits to be shown:

<paths>

Commits modifying the given <paths> are selected.

--simplify-by-decoration

Commits that are referred by some branch or tag are selected.

Note that extra commits can be shown to give a meaningful history.

The following options affect the way the simplification is performed:

Default mode

Simplifies the history to the simplest history explaining the final state of the tree. Simplest because it prunes some side branches if the end result is the same (i.e. merging branches with the same content)

--show-pulls

Include all commits from the default mode, but also any merge commits that are not TREESAME to the first parent but are TREESAME to a later parent. This mode is helpful for showing the merge commits that "first introduced" a change to a branch.

--full-history

Same as the default mode, but does not prune some history.

--dense

Only the selected commits are shown, plus some to have a meaningful history.

--sparse

All commits in the simplified history are shown.

--simplify-merges

Additional option to *--full-history* to remove some needless merges from the resulting history, as there are no selected commits contributing to this merge.

--ancestry-path[=<commit>]

When given a range of commits to display (e.g. *commit1*..*commit2* or *commit2* ^*commit1*), only display commits in that range that are ancestors of <commit>, descendants of <commit>, or <commit> itself. If no commit is specified, use *commit1* (the excluded part of the range) as <commit>. Can be passed multiple times; if so, a commit is included if it is any of the commits given or if it is an ancestor or descendant of one of them.

A more detailed explanation follows.

Suppose you specified *foo* as the <paths>. We shall call commits that modify *foo* !TREESAME, and the rest TREESAME. (In a diff filtered for *foo*, they look different and equal, respectively.)

In the following, we will always refer to the same example history to illustrate the differences between simplification settings. We assume that you are filtering for a file *foo* in this commit graph:

```
          .-A---M---N---O---P---Q
         /     /   /   /   /   /
        I     B   C   D   E   Y
         \   /   /   /   /   /
          `-------------'   X
```

The horizontal line of history A---Q is taken to be the first parent of each merge. The commits are:

- *I* is the initial commit, in which *foo* exists with contents asdf, and a file *quux* exists with contents quux. Initial commits are compared to an empty tree, so *I* is !TREESAME.
- In *A*, *foo* contains just foo.
- *B* contains the same change as *A*. Its merge *M* is trivial and hence TREESAME to all parents.
- *C* does not change *foo*, but its merge *N* changes it to foobar, so it is not TREESAME to any parent.
- *D* sets *foo* to baz. Its merge *O* combines the strings from *N* and *D* to foobarbaz; i.e., it is not TREESAME to any parent.
- *E* changes *quux* to xyzzy, and its merge *P* combines the strings to quux xyzzy. *P* is TREESAME to *O*, but not to *E*.
- *X* is an independent root commit that added a new file *side*, and *Y* modified it. *Y* is TREESAME to *X*. Its merge *Q* added *side* to *P*, and *Q* is TREESAME to *P*, but not to *Y*.

rev-list walks backwards through history, including or excluding commits based on whether --*full-history* and/or parent rewriting (via --*parents* or --*children*) are used. The following settings are available.

Default mode

Commits are included if they are not TREESAME to any parent (though this can be changed, see --*sparse* below). If the commit was a merge, and it was TREESAME to one parent, follow only that parent. (Even if there are several TREESAME parents, follow only one of them.) Otherwise, follow all parents.

This results in:

```
  .-A---N---O
 /     /   /
I---------D
```

Note how the rule to only follow the TREESAME parent, if one is available, removed *B* from consideration entirely. *C* was considered via *N*, but is TREESAME. Root commits are compared to an empty tree, so *I* is !TREESAME.

Parent/child relations are only visible with --*parents*, but that does not affect the commits selected in default mode, so we have shown the parent lines.

--full-history without parent rewriting

This mode differs from the default in one point: always follow all parents of a merge, even if it is TREESAME to one of them. Even if more than one side of the merge has commits that are included, this does not imply that the merge itself is! In the example, we get

```
I  A  B  N  D  O  P  Q
```

M was excluded because it is TREESAME to both parents. *E*, *C* and *B* were all walked, but only *B* was !TREESAME, so the others do not appear.

Note that without parent rewriting, it is not really possible to talk about the parent/child relationships between the commits, so we show them disconnected.

--full-history with parent rewriting

Ordinary commits are only included if they are !TREESAME (though this can be changed, see --*sparse* below).

Merges are always included. However, their parent list is rewritten: Along each parent, prune away commits that are not included themselves. This results in

```
  .-A---M---N---O---P---Q
 /     /   /   /   /
I     B   /   D   /
 \   /   /   /   /
  `-------------'
```

Compare to *--full-history* without rewriting above. Note that *E* was pruned away because it is TREESAME, but the parent list of P was rewritten to contain *E*'s parent *I*. The same happened for *C* and *N*, and *X*, *Y* and *Q*.

In addition to the above settings, you can change whether TREESAME affects inclusion:

--dense

Commits that are walked are included if they are not TREESAME to any parent.

--sparse

All commits that are walked are included.

Note that without *--full-history*, this still simplifies merges: if one of the parents is TREESAME, we follow only that one, so the other sides of the merge are never walked.

--simplify-merges

First, build a history graph in the same way that *--full-history* with parent rewriting does (see above).

Then simplify each commit *C* to its replacement *C'* in the final history according to the following rules:

- Set *C'* to *C*.
- Replace each parent *P* of *C'* with its simplification *P'*. In the process, drop parents that are ancestors of other parents or that are root commits TREESAME to an empty tree, and remove duplicates, but take care to never drop all parents that we are TREESAME to.
- If after this parent rewriting, *C'* is a root or merge commit (has zero or >1 parents), a boundary commit, or !TREESAME, it remains. Otherwise, it is replaced with its only parent.

The effect of this is best shown by way of comparing to *--full-history* with parent rewriting. The example turns into:

```
  .-A---M---N---O
 /     /       /
I     B       D
 \   /       /
  `---------'
```

Note the major differences in *N*, *P*, and *Q* over *--full-history*:

- *N*'s parent list had *I* removed, because it is an ancestor of the other parent *M*. Still, *N* remained because it is !TREESAME.
- *P*'s parent list similarly had *I* removed. *P* was then removed completely, because it had one parent and is TREESAME.
- *Q*'s parent list had *Y* simplified to *X*. *X* was then removed, because it was a TREESAME root. *Q* was then removed completely, because it had one parent and is TREESAME.

There is another simplification mode available:

--ancestry-path[=<commit>]

Limit the displayed commits to those which are an ancestor of <commit>, or which are a descendant of <commit>, or are <commit> itself.

As an example use case, consider the following commit history:

```
          D---E-------F
         /     \       \
        B---C---G---H---I---J
       /                     \
      A-------K---------------L--M
```

A regular *D..M* computes the set of commits that are ancestors of *M*, but excludes the ones that are ancestors of *D*. This is useful to see what happened to the history leading to *M* since *D*, in the sense that what does *M* have that did not exist in *D*. The result in this example would be all the commits, except *A* and *B* (and *D* itself, of course).

When we want to find out what commits in *M* are contaminated with the bug introduced by *D* and need fixing, however, we might want to view only the subset of *D..M* that are actually descendants of *D*, i.e. excluding *C* and *K*. This is exactly what the *--ancestry-path* option does. Applied to the *D..M* range, it results in:

```
E-------F
 \       \
  G---H---I---J
                \
                 L--M
```

We can also use *--ancestry-path=D* instead of *--ancestry-path* which means the same thing when applied to the *D..M* range but is just more explicit.

If we instead are interested in a given topic within this range, and all commits affected by that topic, we may only want to view the subset of *D..M* which contain that topic in their ancestry path. So, using *--ancestry-path=H D..M* for example would result in:

```
E
 \
  G---H---I---J
                \
                 L--M
```

Whereas *--ancestry-path=K D..M* would result in

```
K---------------L--M
```

Before discussing another option, *--show-pulls*, we need to create a new example history.

A common problem users face when looking at simplified history is that a commit they know changed a file somehow does not appear in the file's simplified history. Let's demonstrate a new example and show how options such as *--full-history* and *--simplify-merges* works in that case:

```
     .-A---M-----C--N---O---P
    /     / \  \  \/   /   /
   I     B   \  R-'`-Z'   /
    \   /     \/         /
     \ /      /\        /
      `---X--'   `---Y--'
```

For this example, suppose *I* created *file.txt* which was modified by *A*, *B*, and *X* in different ways. The single-parent commits *C*, *Z*, and *Y* do not change *file.txt*. The merge commit *M* was created by resolving the merge conflict to include both changes from *A* and *B* and hence is not TREESAME to either. The merge commit *R*, however, was created by ignoring the contents of *file.txt* at *M* and taking only the contents of *file.txt* at *X*. Hence, *R* is TREESAME to *X* but not *M*. Finally, the natural merge resolution to create *N* is to take the contents of *file.txt* at *R*, so *N* is TREESAME to *R* but not *C*. The merge commits *O* and *P* are TREESAME to their first parents, but not to their second parents, *Z* and *Y* respectively.

When using the default mode, *N* and *R* both have a TREESAME parent, so those edges are walked and the others are ignored. The resulting history graph is:

```
I---X
```

When using *--full-history*, Git walks every edge. This will discover the commits *A* and *B* and the merge *M*, but also will reveal the merge commits *O* and *P*. With parent rewriting, the resulting graph is:

```
  .-A---M--------N---O---P
 /     / \  \  \/   /   /
I     B   \  R-'`--'   /
 \   /     \/         /
  \ /      /\        /
   `---X--'   `------'
```

Here, the merge commits *O* and *P* contribute extra noise, as they did not actually contribute a change to *file.txt*. They only merged a topic that was based on an older version of *file.txt*. This is a common issue in repositories using a workflow where many contributors work in parallel and merge their topic branches along a single trunk: many unrelated merges appear in the *--full-history* results.

When using the *--simplify-merges* option, the commits *O* and *P* disappear from the results. This is because the rewritten second parents of *O* and *P* are reachable from their first parents. Those edges are removed and then the commits look like single-parent commits that are TREESAME to their parent. This also happens to the commit *N*, resulting in a history view as follows:

```
  .-A---M--.
 /     /    \
I     B      R
 \   /      /
  \ /      /
   `---X--'
```

In this view, we see all of the important single-parent changes from *A*, *B*, and *X*. We also see the carefully-resolved merge *M* and the not-so-carefully-resolved merge *R*. This is usually enough information to determine why the commits *A* and *B* "disappeared" from history in the default view. However, there are a few issues with this approach.

The first issue is performance. Unlike any previous option, the *--simplify-merges* option requires walking the entire commit history before returning a single result. This can make the option difficult to use for very large repositories.

The second issue is one of auditing. When many contributors are working on the same repository, it is important which merge commits introduced a change into an important branch. The problematic merge *R* above is not likely to be the merge commit that was used to merge into an important branch. Instead, the merge *N* was used to merge *R* and *X* into the important branch. This commit may have information about why the change *X* came to override the changes from *A* and *B* in its commit message.

--show-pulls

In addition to the commits shown in the default history, show each merge commit that is not TREESAME to its first parent but is TREESAME to a later parent.

When a merge commit is included by *--show-pulls*, the merge is treated as if it "pulled" the change from another branch. When using *--show-pulls* on this example (and no other options) the resulting graph is:

```
I---X---R---N
```

Here, the merge commits *R* and *N* are included because they pulled the commits *X* and *R* into the base branch, respectively. These merges are the reason the commits *A* and *B* do not appear in the default history.

When *--show-pulls* is paired with *--simplify-merges*, the graph includes all of the necessary information:

```
        .-A---M--.   N
       /     /    \ /
      I     B      R
       \   /      /
        \ /      /
         `---X--'
```

Notice that since *M* is reachable from *R*, the edge from *N* to *M* was simplified away. However, *N* still appears in the history as an important commit because it "pulled" the change *R* into the main branch.

The *--simplify-by-decoration* option allows you to view only the big picture of the topology of the history, by omitting commits that are not referenced by tags. Commits are marked as !TREESAME (in other words, kept after history simplification rules described above) if (1) they are referenced by tags, or (2) they change the contents of the paths given on the command line. All other commits are marked as TREESAME (subject to be simplified away).

3. Commit Ordering

By default, the commits are shown in reverse chronological order.

--date-order

Show no parents before all of its children are shown, but otherwise show commits in the commit timestamp order.

--author-date-order

Show no parents before all of its children are shown, but otherwise show commits in the author timestamp order.

--topo-order

Show no parents before all of its children are shown, and avoid showing commits on multiple lines of history intermixed.

For example, in a commit history like this:

```
---1----2----4----7
    \              \
     3----5----6----8---
```

where the numbers denote the order of commit timestamps, *git rev-list* and friends with *--date-order* show the commits in the timestamp order: 8 7 6 5 4 3 2 1.

With *--topo-order*, they would show 8 6 5 3 7 4 2 1 (or 8 7 4 2 6 5 3 1); some older commits are shown before newer ones in order to avoid showing the commits from two parallel development track mixed together.

--reverse

Output the commits chosen to be shown (see Commit Limiting section above) in reverse order. Cannot be combined with *--walk-reflogs*.

4. Object Traversal

These options are mostly targeted for packing of Git repositories.

--no-walk[=(sorted|unsorted)]

Only show the given commits, but do not traverse their ancestors. This has no effect if a range is specified. If the argument *unsorted* is given, the commits are shown in the order they were given on the command line.

Otherwise (if *sorted* or no argument was given), the commits are shown in reverse chronological order by commit time. Cannot be combined with *--graph*.

--do-walk

Overrides a previous *--no-walk*.

5. Commit Formatting

--pretty[=<format>] , --format=<format>

Pretty-print the contents of the commit logs in a given format, where *<format>* can be one of *oneline*, *short*, *medium*, *full*, *fuller*, *reference*, *email*, *raw*, *format:<string>* and *tformat:<string>*. When *<format>* is none of the above, and has *%placeholder* in it, it acts as if *--pretty=tformat:<format>* were given.

See the "PRETTY FORMATS" section for some additional details for each format. When =*<format>* part is omitted, it defaults to *medium*.

Note: you can specify the default pretty format in the repository configuration (see Section G.3.29, “git-config(1)”).

--abbrev-commit

Instead of showing the full 40-byte hexadecimal commit object name, show a prefix that names the object uniquely. "--abbrev=<n>" (which also modifies diff output, if it is displayed) option can be used to specify the minimum length of the prefix.

This should make "--pretty=oneline" a whole lot more readable for people using 80-column terminals.

--no-abbrev-commit

Show the full 40-byte hexadecimal commit object name. This negates *--abbrev-commit*, either explicit or implied by other options such as "--oneline". It also overrides the *log.abbrevCommit* variable.

--oneline

This is a shorthand for "--pretty=oneline --abbrev-commit" used together.

--encoding=<encoding>

Commit objects record the character encoding used for the log message in their encoding header; this option can be used to tell the command to re-code the commit log message in the encoding preferred by the user. For non plumbing commands this defaults to UTF-8. Note that if an object claims to be encoded in *X* and we are outputting in *X*, we will output the object verbatim; this means that invalid sequences in the original commit may be copied to the output. Likewise, if iconv(3) fails to convert the commit, we will quietly output the original object verbatim.

--expand-tabs=<n> , --expand-tabs , --no-expand-tabs

Perform a tab expansion (replace each tab with enough spaces to fill to the next display column that is a multiple of *<n>*) in the log message before showing it in the output. *--expand-tabs* is a short-hand for *--expand-tabs=8*, and *--no-expand-tabs* is a short-hand for *--expand-tabs=0*, which disables tab expansion.

By default, tabs are expanded in pretty formats that indent the log message by 4 spaces (i.e. *medium*, which is the default, *full*, and *fuller*).

--notes[=<ref>]

Show the notes (see Section G.3.94, “git-notes(1)”) that annotate the commit, when showing the commit log message. This is the default for *git log*, *git show* and *git whatchanged* commands when there is no *--pretty*, *--format*, or *--oneline* option given on the command line.

By default, the notes shown are from the notes refs listed in the *core.notesRef* and *notes.displayRef* variables (or corresponding environment overrides). See Section G.3.29, “git-config(1)” for more details.

With an optional <*ref*> argument, use the ref to find the notes to display. The ref can specify the full refname when it begins with *refs/notes/*; when it begins with *notes/*, *refs/* and otherwise *refs/notes/* is prefixed to form the full name of the ref.

Multiple --notes options can be combined to control which notes are being displayed. Examples: "--notes=foo" will show only notes from "refs/notes/foo"; "--notes=foo --notes" will show both notes from "refs/notes/foo" and from the default notes ref(s).

--no-notes

Do not show notes. This negates the above *--notes* option, by resetting the list of notes refs from which notes are shown. Options are parsed in the order given on the command line, so e.g. "--notes --notes=foo --no-notes --notes=bar" will only show notes from "refs/notes/bar".

--show-notes-by-default

Show the default notes unless options for displaying specific notes are given.

--show-notes[=<ref>] , --[no-]standard-notes

These options are deprecated. Use the above --notes/--no-notes options instead.

--show-signature

Check the validity of a signed commit object by passing the signature to *gpg --verify* and show the output.

--relative-date

Synonym for *--date=relative*.

--date=<format>

Only takes effect for dates shown in human-readable format, such as when using *--pretty*. *log.date* config variable sets a default value for the log command's *--date* option. By default, dates are shown in the original time zone (either committer's or author's). If *-local* is appended to the format (e.g., *iso-local*), the user's local time zone is used instead.

--date=relative shows dates relative to the current time, e.g. 2 hours ago. The *-local* option has no effect for *--date=relative*.

--date=local is an alias for *--date=default-local*.

--date=iso (or *--date=iso8601*) shows timestamps in a ISO 8601-like format. The differences to the strict ISO 8601 format are:

- a space instead of the *T* date/time delimiter
- a space between time and time zone
- no colon between hours and minutes of the time zone

--date=iso-strict (or *--date=iso8601-strict*) shows timestamps in strict ISO 8601 format.

--date=rfc (or *--date=rfc2822*) shows timestamps in RFC 2822 format, often found in email messages.

--date=short shows only the date, but not the time, in *YYYY-MM-DD* format.

--date=raw shows the date as seconds since the epoch (1970-01-01 00:00:00 UTC), followed by a space, and then the timezone as an offset from UTC (a + or - with four digits; the first two are hours, and the

second two are minutes). I.e., as if the timestamp were formatted with *strftime("%s %z")*). Note that the *-local* option does not affect the seconds-since-epoch value (which is always measured in UTC), but does switch the accompanying timezone value.

--date=human shows the timezone if the timezone does not match the current time-zone, and doesn't print the whole date if that matches (ie skip printing year for dates that are "this year", but also skip the whole date itself if it's in the last few days and we can just say what weekday it was). For older dates the hour and minute is also omitted.

--date=unix shows the date as a Unix epoch timestamp (seconds since 1970). As with *--raw*, this is always in UTC and therefore *-local* has no effect.

--date=format:... feeds the format ... to your system *strftime*, except for %s, %z, and %Z, which are handled internally. Use *--date=format:%c* to show the date in your system locale's preferred format. See the *strftime* manual for a complete list of format placeholders. When using *-local*, the correct syntax is *--date=format-local:...*.

--date=default is the default format, and is based on ctime(3) output. It shows a single line with three-letter day of the week, three-letter month, day-of-month, hour-minute-seconds in "HH:MM:SS" format, followed by 4-digit year, plus timezone information, unless the local time zone is used, e.g. *Thu Jan 1 00:00:00 1970 +0000*.

--parents

Print also the parents of the commit (in the form "commit parent…"). Also enables parent rewriting, see *History Simplification* above.

--children

Print also the children of the commit (in the form "commit child…"). Also enables parent rewriting, see *History Simplification* above.

--left-right

Mark which side of a symmetric difference a commit is reachable from. Commits from the left side are prefixed with < and those from the right with >. If combined with *--boundary*, those commits are prefixed with -.

For example, if you have this topology:

```
             y---b---b  branch B
            / \ /
           /   .
          /   / \
         o---x---a---a  branch A
```

you would get an output like this:

```
$ git rev-list --left-right --boundary --pretty=oneline A...B

>bbbbbbb... 3rd on b
>bbbbbbb... 2nd on b
<aaaaaaa... 3rd on a
<aaaaaaa... 2nd on a
-yyyyyyy... 1st on b
-xxxxxxx... 1st on a
```

--graph

Draw a text-based graphical representation of the commit history on the left hand side of the output. This may cause extra lines to be printed in between commits, in order for the graph history to be drawn properly. Cannot be combined with *--no-walk*.

This enables parent rewriting, see *History Simplification* above.

This implies the *--topo-order* option by default, but the *--date-order* option may also be specified.

--show-linear-break[=<barrier>]

When --graph is not used, all history branches are flattened which can make it hard to see that the two consecutive commits do not belong to a linear branch. This option puts a barrier in between them in that case. If *<barrier>* is specified, it is the string that will be shown instead of the default one.

OUTPUT

When there are no conflicts, the output of this command is usable as input to *git update-ref --stdin*. It is of the form:

```
update refs/heads/branch1 ${NEW_branch1_HASH} ${OLD_branch1_HASH}
update refs/heads/branch2 ${NEW_branch2_HASH} ${OLD_branch2_HASH}
update refs/heads/branch3 ${NEW_branch3_HASH} ${OLD_branch3_HASH}
```

where the number of refs updated depends on the arguments passed and the shape of the history being replayed. When using *--advance*, the number of refs updated is always one, but for *--onto*, it can be one or more (rebasing multiple branches simultaneously is supported).

EXIT STATUS

For a successful, non-conflicted replay, the exit status is 0. When the replay has conflicts, the exit status is 1. If the replay is not able to complete (or start) due to some kind of error, the exit status is something other than 0 or 1.

EXAMPLES

To simply rebase *mybranch* onto *target*:

```
$ git replay --onto target origin/main..mybranch
update refs/heads/mybranch ${NEW_mybranch_HASH} ${OLD_mybranch_HASH}
```

To cherry-pick the commits from mybranch onto target:

```
$ git replay --advance target origin/main..mybranch
update refs/heads/target ${NEW_target_HASH} ${OLD_target_HASH}
```

Note that the first two examples replay the exact same commits and on top of the exact same new base, they only differ in that the first provides instructions to make mybranch point at the new commits and the second provides instructions to make target point at them.

What if you have a stack of branches, one depending upon another, and you'd really like to rebase the whole set?

```
$ git replay --contained --onto origin/main origin/main..tipbranch
update refs/heads/branch1 ${NEW_branch1_HASH} ${OLD_branch1_HASH}
update refs/heads/branch2 ${NEW_branch2_HASH} ${OLD_branch2_HASH}
update refs/heads/tipbranch ${NEW_tipbranch_HASH} ${OLD_tipbranch_HASH}
```

When calling *git replay*, one does not need to specify a range of commits to replay using the syntax *A..B*; any range expression will do:

```
$ git replay --onto origin/main ^base branch1 branch2 branch3
update refs/heads/branch1 ${NEW_branch1_HASH} ${OLD_branch1_HASH}
update refs/heads/branch2 ${NEW_branch2_HASH} ${OLD_branch2_HASH}
update refs/heads/branch3 ${NEW_branch3_HASH} ${OLD_branch3_HASH}
```

This will simultaneously rebase *branch1*, *branch2*, and *branch3*, all commits they have since *base*, playing them on top of *origin/main*. These three branches may have commits on top of *base* that they have in common, but that does not need to be the case.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.116. git-request-pull(1)

2024-07-02
Revision History
2024-07-02

NAME

git-request-pull - Generates a summary of pending changes

SYNOPSIS

git request-pull [-p] <start> <URL> [<end>]

DESCRIPTION

Generate a request asking your upstream project to pull changes into their tree. The request, printed to the standard output, begins with the branch description, summarizes the changes, and indicates from where they can be pulled.

The upstream project is expected to have the commit named by *<start>* and the output asks it to integrate the changes you made since that commit, up to the commit named by *<end>*, by visiting the repository named by *<URL>*.

OPTIONS

-p

Include patch text in the output.

<start>

Commit to start at. This names a commit that is already in the upstream history.

<URL>

The repository URL to be pulled from.

<end>

Commit to end at (defaults to HEAD). This names the commit at the tip of the history you are asking to be pulled.

When the repository named by *<URL>* has the commit at a tip of a ref that is different from the ref you have locally, you can use the *<local>:<remote>* syntax, to have its local name, a colon *:*, and its remote name.

EXAMPLES

Imagine that you built your work on your *master* branch on top of the *v1.0* release, and want it to be integrated into the project. First you push that change to your public repository for others to see:

```
git push https://git.ko.xz/project master
```

Then, you run this command:

```
git request-pull v1.0 https://git.ko.xz/project master
```

which will produce a request to the upstream, summarizing the changes between the *v1.0* release and your *master*, to pull it from your public repository.

If you pushed your change to a branch whose name is different from the one you have locally, e.g.

```
git push https://git.ko.xz/project master:for-linus
```

then you can ask that to be pulled with

```
git request-pull v1.0 https://git.ko.xz/project master:for-linus
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.117. git-rerere(1)

2024-07-02
Revision History
2024-07-02

NAME

git-rerere - Reuse recorded resolution of conflicted merges

SYNOPSIS

git rerere [clear | forget <pathspec>… | diff | status | remaining | gc]

DESCRIPTION

In a workflow employing relatively long lived topic branches, the developer sometimes needs to resolve the same conflicts over and over again until the topic branches are done (either merged to the "release" branch, or sent out and accepted upstream).

This command assists the developer in this process by recording conflicted automerge results and corresponding hand resolve results on the initial manual merge, and applying previously recorded hand resolutions to their corresponding automerge results.

> **Note**
>
> You need to set the configuration variable *rerere.enabled* in order to enable this command.

COMMANDS

Normally, *git rerere* is run without arguments or user-intervention. However, it has several commands that allow it to interact with its working state.

clear

Reset the metadata used by rerere if a merge resolution is to be aborted. Calling *git am [--skip|--abort]* or *git rebase [--skip|--abort]* will automatically invoke this command.

forget <pathspec>

Reset the conflict resolutions which rerere has recorded for the current conflict in <pathspec>.

diff

Display diffs for the current state of the resolution. It is useful for tracking what has changed while the user is resolving conflicts. Additional arguments are passed directly to the system *diff* command installed in PATH.

status

Print paths with conflicts whose merge resolution rerere will record.

remaining

Print paths with conflicts that have not been autoresolved by rerere. This includes paths whose resolutions cannot be tracked by rerere, such as conflicting submodules.

gc

Prune records of conflicted merges that occurred a long time ago. By default, unresolved conflicts older than 15 days and resolved conflicts older than 60 days are pruned. These defaults are controlled via the *gc.rerereUnresolved* and *gc.rerereResolved* configuration variables respectively.

DISCUSSION

When your topic branch modifies an overlapping area that your master branch (or upstream) touched since your topic branch forked from it, you may want to test it with the latest master, even before your topic branch is ready to be pushed upstream:

```
            o---*---o topic
           /
o---o---o---*---o---o master
```

For such a test, you need to merge master and topic somehow. One way to do it is to pull master into the topic branch:

```
$ git switch topic
$ git merge master

            o---*---o---+ topic
           /           /
o---o---o---*---o---o master
```

The commits marked with * touch the same area in the same file; you need to resolve the conflicts when creating the commit marked with +. Then you can test the result to make sure your work-in-progress still works with what is in the latest master.

After this test merge, there are two ways to continue your work on the topic. The easiest is to build on top of the test merge commit +, and when your work in the topic branch is finally ready, pull the topic branch into master, and/or ask the upstream to pull from you. By that time, however, the master or the upstream might have been advanced since the test merge +, in which case the final commit graph would look like this:

```
$ git switch topic
$ git merge master
$ ... work on both topic and master branches
$ git switch master
$ git merge topic

            o---*---o---+---o---o topic
           /           /         \
o---o---o---*---o---o---o---o---+ master
```

When your topic branch is long-lived, however, your topic branch would end up having many such "Merge from master" commits on it, which would unnecessarily clutter the development history. Readers of the Linux kernel mailing list may remember that Linus complained about such too frequent test merges when a subsystem maintainer asked to pull from a branch full of "useless merges".

As an alternative, to keep the topic branch clean of test merges, you could blow away the test merge, and keep building on top of the tip before the test merge:

```
$ git switch topic
$ git merge master
$ git reset --hard HEAD^ ;# rewind the test merge
```

```
$ ... work on both topic and master branches
$ git switch master
$ git merge topic

      o---*---o-------o---o topic
     /                     \
o---o---o---*---o---o---o---o---+ master
```

This would leave only one merge commit when your topic branch is finally ready and merged into the master branch. This merge would require you to resolve the conflict, introduced by the commits marked with *. However, this conflict is often the same conflict you resolved when you created the test merge you blew away. *git rerere* helps you resolve this final conflicted merge using the information from your earlier hand resolve.

Running the *git rerere* command immediately after a conflicted automerge records the conflicted working tree files, with the usual conflict markers `<<<<<<<`, `=======`, and `>>>>>>>` in them. Later, after you are done resolving the conflicts, running *git rerere* again will record the resolved state of these files. Suppose you did this when you created the test merge of master into the topic branch.

Next time, after seeing the same conflicted automerge, running *git rerere* will perform a three-way merge between the earlier conflicted automerge, the earlier manual resolution, and the current conflicted automerge. If this three-way merge resolves cleanly, the result is written out to your working tree file, so you do not have to manually resolve it. Note that *git rerere* leaves the index file alone, so you still need to do the final sanity checks with *git diff* (or *git diff -c*) and *git add* when you are satisfied.

As a convenience measure, *git merge* automatically invokes *git rerere* upon exiting with a failed automerge and *git rerere* records the hand resolve when it is a new conflict, or reuses the earlier hand resolve when it is not. *git commit* also invokes *git rerere* when committing a merge result. What this means is that you do not have to do anything special yourself (besides enabling the rerere.enabled config variable).

In our example, when you do the test merge, the manual resolution is recorded, and it will be reused when you do the actual merge later with the updated master and topic branch, as long as the recorded resolution is still applicable.

The information *git rerere* records is also used when running *git rebase*. After blowing away the test merge and continuing development on the topic branch:

```
      o---*---o-------o---o topic
     /
o---o---o---*---o---o---o---o   master

$ git rebase master topic

                                o---*---o-------o---o topic
                               /
o---o---o---*---o---o---o---o   master
```

you could run *git rebase master topic*, to bring yourself up to date before your topic is ready to be sent upstream. This would result in falling back to a three-way merge, and it would conflict the same way as the test merge you resolved earlier. *git rerere* will be run by *git rebase* to help you resolve this conflict.

[NOTE] *git rerere* relies on the conflict markers in the file to detect the conflict. If the file already contains lines that look the same as lines with conflict markers, *git rerere* may fail to record a conflict resolution. To work around this, the *conflict-marker-size* setting in Section G.4.2, “gitattributes(5)” can be used.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.118. git-reset(1)

2024-07-02
Revision History

2024-07-02

NAME

git-reset - Reset current HEAD to the specified state

SYNOPSIS

git reset [-q] [<tree-ish>] [--] <pathspec>…
git reset [-q] [--pathspec-from-file=<file> [--pathspec-file-nul]] [<tree-ish>]
git reset (--patch | -p) [<tree-ish>] [--] [<pathspec>…]
git reset [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]

DESCRIPTION

In the first three forms, copy entries from <*tree-ish*> to the index. In the last form, set the current branch head (*HEAD*) to <*commit*>, optionally modifying index and working tree to match. The <*tree-ish*>/<*commit*> defaults to *HEAD* in all forms.

git reset [-q] [<tree-ish>] [--] <pathspec>… , *git reset* [-q] [--pathspec-from-file=<file> [--pathspec-file-nul]] [<tree-ish>]

These forms reset the index entries for all paths that match the <*pathspec*> to their state at <*tree-ish*>. (It does not affect the working tree or the current branch.)

This means that *git reset* <*pathspec*> is the opposite of *git add* <*pathspec*>. This command is equivalent to *git restore [--source=<tree-ish>] --staged* <*pathspec*>....

After running *git reset* <*pathspec*> to update the index entry, you can use Section G.3.119, “git-restore(1)” to check the contents out of the index to the working tree. Alternatively, using Section G.3.119, “git-restore(1)” and specifying a commit with *--source*, you can copy the contents of a path out of a commit to the index and to the working tree in one go.

git reset (--patch | -p) [<tree-ish>] [--] [<pathspec>…]

Interactively select hunks in the difference between the index and <*tree-ish*> (defaults to *HEAD*). The chosen hunks are applied in reverse to the index.

This means that *git reset -p* is the opposite of *git add -p*, i.e. you can use it to selectively reset hunks. See the Interactive Mode section of Section G.3.2, “git-add(1)” to learn how to operate the *--patch* mode.

git reset [<mode>] [<commit>]

This form resets the current branch head to <*commit*> and possibly updates the index (resetting it to the tree of <*commit*>) and the working tree depending on <*mode*>. Before the operation, *ORIG_HEAD* is set to the tip of the current branch. If <*mode*> is omitted, defaults to *--mixed*. The <*mode*> must be one of the following:

--soft

Does not touch the index file or the working tree at all (but resets the head to <*commit*>, just like all modes do). This leaves all your changed files "Changes to be committed", as *git status* would put it.

--mixed

Resets the index but not the working tree (i.e., the changed files are preserved but not marked for commit) and reports what has not been updated. This is the default action.

If *-N* is specified, removed paths are marked as intent-to-add (see Section G.3.2, “git-add(1)”).

--hard

Resets the index and working tree. Any changes to tracked files in the working tree since <*commit*> are discarded. Any untracked files or directories in the way of writing any tracked files are simply deleted.

--merge

Resets the index and updates the files in the working tree that are different between <*commit*> and *HEAD*, but keeps those which are different between the index and working tree (i.e. which have changes which have not been added). If a file that is different between <*commit*> and the index has unstaged changes, reset is aborted.

In other words, *--merge* does something like a *git read-tree -u -m* <*commit*>, but carries forward unmerged index entries.

--keep

Resets index entries and updates files in the working tree that are different between <*commit*> and *HEAD*. If a file that is different between <*commit*> and *HEAD* has local changes, reset is aborted.

--[no-]recurse-submodules

When the working tree is updated, using --recurse-submodules will also recursively reset the working tree of all active submodules according to the commit recorded in the superproject, also setting the submodules' HEAD to be detached at that commit.

See "Reset, restore and revert" in Section G.3.1, “git(1)” for the differences between the three commands.

OPTIONS

-q , --quiet

Be quiet, only report errors.

--refresh , --no-refresh

Refresh the index after a mixed reset. Enabled by default.

--pathspec-from-file=<file>

Pathspec is passed in <*file*> instead of commandline args. If <*file*> is exactly - then standard input is used. Pathspec elements are separated by LF or CR/LF. Pathspec elements can be quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). See also *--pathspec-file-nul* and global *--literal-pathspecs*.

--pathspec-file-nul

Only meaningful with *--pathspec-from-file*. Pathspec elements are separated with NUL character and all other characters are taken literally (including newlines and quotes).

--

Do not interpret any more arguments as options.

<pathspec>…

Limits the paths affected by the operation.

For more details, see the *pathspec* entry in Section G.4.19, “gitglossary(7)”.

EXAMPLES

Undo add

```
$ edit                                  ❶
$ git add frotz.c filfre.c
```

```
$ mailx                                  ❷
$ git reset                              ❸
$ git pull git://info.example.com/ nitfol  ❹
```

❶ You are happily working on something, and find the changes in these files are in good order. You do not want to see them when you run *git diff*, because you plan to work on other files and changes with these files are distracting.

❷ Somebody asks you to pull, and the changes sound worthy of merging.

❸ However, you already dirtied the index (i.e. your index does not match the *HEAD* commit). But you know the pull you are going to make does not affect *frotz.c* or *filfre.c*, so you revert the index changes for these two files. Your changes in working tree remain there.

❹ Then you can pull and merge, leaving *frotz.c* and *filfre.c* changes still in the working tree.

Undo a commit and redo

```
$ git commit ...
$ git reset --soft HEAD^      ❶
$ edit                        ❷
$ git commit -a -c ORIG_HEAD  ❸
```

❶ This is most often done when you remembered what you just committed is incomplete, or you misspelled your commit message, or both. Leaves working tree as it was before "reset".

❷ Make corrections to working tree files.

❸ "reset" copies the old head to *.git/ORIG_HEAD*; redo the commit by starting with its log message. If you do not need to edit the message further, you can give *-C* option instead.

See also the *--amend* option to Section G.3.28, “git-commit(1)”.

Undo a commit, making it a topic branch

```
$ git branch topic/wip          ❶
$ git reset --hard HEAD~3       ❷
$ git switch topic/wip          ❸
```

❶ You have made some commits, but realize they were premature to be in the *master* branch. You want to continue polishing them in a topic branch, so create *topic/wip* branch off of the current *HEAD*.

❷ Rewind the master branch to get rid of those three commits.

❸ Switch to *topic/wip* branch and keep working.

Undo commits permanently

```
$ git commit ...
$ git reset --hard HEAD~3   ❶
```

❶ The last three commits (*HEAD*, *HEAD^*, and *HEAD~2*) were bad and you do not want to ever see them again. Do **not** do this if you have already given these commits to somebody else. (See the "RECOVERING FROM UPSTREAM REBASE" section in Section G.3.107, “git-rebase(1)” for the implications of doing so.)

Undo a merge or pull

```
$ git pull                         ❶
Auto-merging nitfol
CONFLICT (content): Merge conflict in nitfol
Automatic merge failed; fix conflicts and then commit the result.
$ git reset --hard                 ❷
$ git pull . topic/branch          ❸
Updating from 41223... to 13134...
Fast-forward
```

```
$ git reset --hard ORIG_HEAD        ❹
```

❶ Try to update from the upstream resulted in a lot of conflicts; you were not ready to spend a lot of time merging right now, so you decide to do that later.
❷ "pull" has not made merge commit, so *git reset --hard* which is a synonym for *git reset --hard HEAD* clears the mess from the index file and the working tree.
❸ Merge a topic branch into the current branch, which resulted in a fast-forward.
❹ But you decided that the topic branch is not ready for public consumption yet. "pull" or "merge" always leaves the original tip of the current branch in *ORIG_HEAD*, so resetting hard to it brings your index file and the working tree back to that state, and resets the tip of the branch to that commit.

Undo a merge or pull inside a dirty working tree

```
$ git pull                          ❶
Auto-merging nitfol
Merge made by recursive.
 nitfol                |   20 +++++----
 ...
$ git reset --merge ORIG_HEAD       ❷
```

❶ Even if you may have local modifications in your working tree, you can safely say *git pull* when you know that the change in the other branch does not overlap with them.
❷ After inspecting the result of the merge, you may find that the change in the other branch is unsatisfactory. Running *git reset --hard ORIG_HEAD* will let you go back to where you were, but it will discard your local changes, which you do not want. *git reset --merge* keeps your local changes.

Interrupted workflow

Suppose you are interrupted by an urgent fix request while you are in the middle of a large change. The files in your working tree are not in any shape to be committed yet, but you need to get to the other branch for a quick bugfix.

```
$ git switch feature  ;# you were working in "feature" branch and
$ work work work      ;# got interrupted
$ git commit -a -m "snapshot WIP"                 ❶
$ git switch master
$ fix fix fix
$ git commit ;# commit with real log
$ git switch feature
$ git reset --soft HEAD^ ;# go back to WIP state  ❷
$ git reset                                       ❸
```

❶ This commit will get blown away so a throw-away log message is OK.
❷ This removes the *WIP* commit from the commit history, and sets your working tree to the state just before you made that snapshot.
❸ At this point the index file still has all the WIP changes you committed as *snapshot WIP*. This updates the index to show your WIP files as uncommitted.

See also Section G.3.137, “git-stash(1)”.

Reset a single file in the index

Suppose you have added a file to your index, but later decide you do not want to add it to your commit. You can remove the file from the index while keeping your changes with git reset.

```
$ git reset -- frotz.c                      ❶
$ git commit -m "Commit files in index"     ❷
$ git add frotz.c                           ❸
```

❶ This removes the file from the index while keeping it in the working directory.

❷ This commits all other changes in the index.
❸ Adds the file to the index again.

Keep changes in working tree while discarding some previous commits

Suppose you are working on something and you commit it, and then you continue working a bit more, but now you think that what you have in your working tree should be in another branch that has nothing to do with what you committed previously. You can start a new branch and reset it while keeping the changes in your working tree.

```
$ git tag start
$ git switch -c branch1
$ edit
$ git commit ...                    ❶
$ edit
$ git switch -c branch2             ❷
$ git reset --keep start            ❸
```

❶ This commits your first edits in *branch1*.
❷ In the ideal world, you could have realized that the earlier commit did not belong to the new topic when you created and switched to *branch2* (i.e. *git switch -c branch2 start*), but nobody is perfect.
❸ But you can use *reset --keep* to remove the unwanted commit after you switched to *branch2*.

Split a commit apart into a sequence of commits

Suppose that you have created lots of logically separate changes and committed them together. Then, later you decide that it might be better to have each logical chunk associated with its own commit. You can use git reset to rewind history without changing the contents of your local files, and then successively use *git add -p* to interactively select which hunks to include into each commit, using *git commit -c* to pre-populate the commit message.

```
$ git reset -N HEAD^                ❶
$ git add -p                        ❷
$ git diff --cached                 ❸
$ git commit -c HEAD@{1}            ❹
...                                 ❺
$ git add ...                       ❻
$ git diff --cached                 ❼
$ git commit ...                    ❽
```

❶ First, reset the history back one commit so that we remove the original commit, but leave the working tree with all the changes. The -N ensures that any new files added with *HEAD* are still marked so that *git add -p* will find them.
❷ Next, we interactively select diff hunks to add using the *git add -p* facility. This will ask you about each diff hunk in sequence and you can use simple commands such as "yes, include this", "No don't include this" or even the very powerful "edit" facility.
❸ Once satisfied with the hunks you want to include, you should verify what has been prepared for the first commit by using *git diff --cached*. This shows all the changes that have been moved into the index and are about to be committed.
❹ Next, commit the changes stored in the index. The *-c* option specifies to pre-populate the commit message from the original message that you started with in the first commit. This is helpful to avoid retyping it. The *HEAD@{1}* is a special notation for the commit that *HEAD* used to be at prior to the original reset commit (1 change ago). See Section G.3.109, “git-reflog(1)” for more details. You may also use any other valid commit reference.
❺ You can repeat steps 2-4 multiple times to break the original code into any number of commits.
❻ Now you've split out many of the changes into their own commits, and might no longer use the patch mode of *git add*, in order to select all remaining uncommitted changes.
❼ Once again, check to verify that you've included what you want to. You may also wish to verify that git diff doesn't show any remaining changes to be committed later.

❽ And finally create the final commit.

DISCUSSION

The tables below show what happens when running:

```
git reset --option target
```

to reset the *HEAD* to another commit (*target*) with the different reset options depending on the state of the files.

In these tables, *A*, *B*, *C* and *D* are some different states of a file. For example, the first line of the first table means that if a file is in state *A* in the working tree, in state *B* in the index, in state *C* in *HEAD* and in state *D* in the target, then *git reset --soft target* will leave the file in the working tree in state *A* and in the index in state *B*. It resets (i.e. moves) the *HEAD* (i.e. the tip of the current branch, if you are on one) to *target* (which has the file in state *D*).

```
working index HEAD target         working index HEAD
----------------------------------------------------
 A       B     C    D     --soft   A       B     D
                          --mixed  A       D     D
                          --hard   D       D     D
                          --merge (disallowed)
                          --keep  (disallowed)

working index HEAD target         working index HEAD
----------------------------------------------------
 A       B     C    C     --soft   A       B     C
                          --mixed  A       C     C
                          --hard   C       C     C
                          --merge (disallowed)
                          --keep   A       C     C

working index HEAD target         working index HEAD
----------------------------------------------------
 B       B     C    D     --soft   B       B     D
                          --mixed  B       D     D
                          --hard   D       D     D
                          --merge  D       D     D
                          --keep  (disallowed)

working index HEAD target         working index HEAD
----------------------------------------------------
 B       B     C    C     --soft   B       B     C
                          --mixed  B       C     C
                          --hard   C       C     C
                          --merge  C       C     C
                          --keep   B       C     C

working index HEAD target         working index HEAD
----------------------------------------------------
 B       C     C    D     --soft   B       C     D
                          --mixed  B       D     D
                          --hard   D       D     D
                          --merge (disallowed)
                          --keep  (disallowed)

working index HEAD target         working index HEAD
----------------------------------------------------
 B       C     C    C     --soft   B       C     C
                          --mixed  B       C     C
                          --hard   C       C     C
```

```
                                --merge  B       C     C
                                --keep   B       C     C
```

reset --merge is meant to be used when resetting out of a conflicted merge. Any mergy operation guarantees that the working tree file that is involved in the merge does not have a local change with respect to the index before it starts, and that it writes the result out to the working tree. So if we see some difference between the index and the target and also between the index and the working tree, then it means that we are not resetting out from a state that a mergy operation left after failing with a conflict. That is why we disallow *--merge* option in this case.

reset --keep is meant to be used when removing some of the last commits in the current branch while keeping changes in the working tree. If there could be conflicts between the changes in the commit we want to remove and the changes in the working tree we want to keep, the reset is disallowed. That's why it is disallowed if there are both changes between the working tree and *HEAD*, and between *HEAD* and the target. To be safe, it is also disallowed when there are unmerged entries.

The following tables show what happens when there are unmerged entries:

```
working index HEAD target         working index HEAD
----------------------------------------------------
 X       U     A    B     --soft  (disallowed)
                          --mixed  X       B     B
                          --hard   B       B     B
                          --merge  B       B     B
                          --keep  (disallowed)

working index HEAD target         working index HEAD
----------------------------------------------------
 X       U     A    A     --soft  (disallowed)
                          --mixed  X       A     A
                          --hard   A       A     A
                          --merge  A       A     A
                          --keep  (disallowed)
```

X means any state and *U* means an unmerged index.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.119. git-restore(1)

2024-07-02

Revision History

2024-07-02

NAME

git-restore - Restore working tree files

SYNOPSIS

git restore [<options>] [--source=<tree>] [--staged] [--worktree] [--] <pathspec>…
git restore [<options>] [--source=<tree>] [--staged] [--worktree] --pathspec-from-file=<file> [--pathspec-file-nul]
git restore (-p|--patch) [<options>] [--source=<tree>] [--staged] [--worktree] [--] [<pathspec>…]

DESCRIPTION

Restore specified paths in the working tree with some contents from a restore source. If a path is tracked but does not exist in the restore source, it will be removed to match the source.

The command can also be used to restore the content in the index with *--staged*, or restore both the working tree and the index with *--staged --worktree*.

By default, if *--staged* is given, the contents are restored from *HEAD*, otherwise from the index. Use *--source* to restore from a different commit.

See "Reset, restore and revert" in Section G.3.1, “git(1)” for the differences between the three commands.

THIS COMMAND IS EXPERIMENTAL. THE BEHAVIOR MAY CHANGE.

OPTIONS

-s <tree> , --source=<tree>

Restore the working tree files with the content from the given tree. It is common to specify the source tree by naming a commit, branch or tag associated with it.

If not specified, the contents are restored from *HEAD* if *--staged* is given, otherwise from the index.

As a special case, you may use *"A...B"* as a shortcut for the merge base of *A* and *B* if there is exactly one merge base. You can leave out at most one of *A* and *B*, in which case it defaults to *HEAD*.

-p , --patch

Interactively select hunks in the difference between the restore source and the restore location. See the Interactive Mode section of Section G.3.2, “git-add(1)” to learn how to operate the *--patch* mode.

Note that *--patch* can accept no pathspec and will prompt to restore all modified paths.

-W , --worktree , -S , --staged

Specify the restore location. If neither option is specified, by default the working tree is restored. Specifying *--staged* will only restore the index. Specifying both restores both.

-q , --quiet

Quiet, suppress feedback messages. Implies *--no-progress*.

--progress , --no-progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless *--quiet* is specified. This flag enables progress reporting even if not attached to a terminal, regardless of *--quiet*.

--ours , --theirs

When restoring files in the working tree from the index, use stage #2 (*ours*) or #3 (*theirs*) for unmerged paths. This option cannot be used when checking out paths from a tree-ish (i.e. with the *--source* option).

Note that during *git rebase* and *git pull --rebase*, *ours* and *theirs* may appear swapped. See the explanation of the same options in Section G.3.19, “git-checkout(1)” for details.

-m , --merge

When restoring files on the working tree from the index, recreate the conflicted merge in the unmerged paths. This option cannot be used when checking out paths from a tree-ish (i.e. with the *--source* option).

--conflict=<style>

The same as *--merge* option above, but changes the way the conflicting hunks are presented, overriding the *merge.conflictStyle* configuration variable. Possible values are "merge" (default), "diff3", and "zdiff3".

--ignore-unmerged

When restoring files on the working tree from the index, do not abort the operation if there are unmerged entries and neither *--ours*, *--theirs*, *--merge* or *--conflict* is specified. Unmerged paths on the working tree are left alone.

--ignore-skip-worktree-bits

In sparse checkout mode, the default is to only update entries matched by <*pathspec*> and sparse patterns in $GIT_DIR/info/sparse-checkout. This option ignores the sparse patterns and unconditionally restores any files in <*pathspec*>.

--recurse-submodules , --no-recurse-submodules

If <*pathspec*> names an active submodule and the restore location includes the working tree, the submodule will only be updated if this option is given, in which case its working tree will be restored to the commit recorded in the superproject, and any local modifications overwritten. If nothing (or *--no-recurse-submodules*) is used, submodules working trees will not be updated. Just like Section G.3.19, "git-checkout(1)", this will detach *HEAD* of the submodule.

--overlay , --no-overlay

In overlay mode, the command never removes files when restoring. In no-overlay mode, tracked files that do not appear in the *--source* tree are removed, to make them match <*tree*> exactly. The default is no-overlay mode.

--pathspec-from-file=<file>

Pathspec is passed in <*file*> instead of commandline args. If <*file*> is exactly - then standard input is used. Pathspec elements are separated by LF or CR/LF. Pathspec elements can be quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, "git-config(1)"). See also *--pathspec-file-nul* and global *--literal-pathspecs*.

--pathspec-file-nul

Only meaningful with *--pathspec-from-file*. Pathspec elements are separated with NUL character and all other characters are taken literally (including newlines and quotes).

--

Do not interpret any more arguments as options.

<pathspec>…

Limits the paths affected by the operation.

For more details, see the *pathspec* entry in Section G.4.19, "gitglossary(7)".

EXAMPLES

The following sequence switches to the *master* branch, reverts the *Makefile* to two revisions back, deletes hello.c by mistake, and gets it back from the index.

```
$ git switch master
$ git restore --source master~2 Makefile  ❶
$ rm -f hello.c
$ git restore hello.c                     ❷
```

❶ take a file out of another commit
❷ restore hello.c from the index

If you want to restore *all* C source files to match the version in the index, you can say

```
$ git restore '*.c'
```

Note the quotes around **.c*. The file *hello.c* will also be restored, even though it is no longer in the working tree, because the file globbing is used to match entries in the index (not in the working tree by the shell).

To restore all files in the current directory

```
$ git restore .
```

or to restore all working tree files with *top* pathspec magic (see Section G.4.19, “gitglossary(7)”)

```
$ git restore :/
```

To restore a file in the index to match the version in *HEAD* (this is the same as using Section G.3.118, “git-reset(1)”)

```
$ git restore --staged hello.c
```

or you can restore both the index and the working tree (this is the same as using Section G.3.19, “git-checkout(1)”)

```
$ git restore --source=HEAD --staged --worktree hello.c
```

or the short form which is more practical but less readable:

```
$ git restore -s@ -SW hello.c
```

SEE ALSO

Section G.3.19, “git-checkout(1)”, Section G.3.118, “git-reset(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.120. git-rev-list(1)

2024-07-02
Revision History
2024-07-02

NAME

git-rev-list - Lists commit objects in reverse chronological order

SYNOPSIS

git rev-list [<options>] <commit>… [--] [<path>…]

DESCRIPTION

List commits that are reachable by following the *parent* links from the given commit(s), but exclude commits that are reachable from the one(s) given with a ^ in front of them. The output is given in reverse chronological order by default.

You can think of this as a set operation. Commits reachable from any of the commits given on the command line form a set, and then commits reachable from any of the ones given with ^ in front are subtracted from that set. The remaining commits are what comes out in the command's output. Various other options and paths parameters can be used to further limit the result.

Thus, the following command:

```
$ git rev-list foo bar ^baz
```

means "list all the commits which are reachable from *foo* or *bar*, but not from *baz*".

A special notation "*<commit1>..<commit2>*" can be used as a short-hand for "^*<commit1>* *<commit2>*". For example, either of the following may be used interchangeably:

```
$ git rev-list origin..HEAD
```

```
$ git rev-list HEAD ^origin
```

Another special notation is "*<commit1>…<commit2>*" which is useful for merges. The resulting set of commits is the symmetric difference between the two operands. The following two commands are equivalent:

```
$ git rev-list A B --not $(git merge-base --all A B)
$ git rev-list A...B
```

rev-list is an essential Git command, since it provides the ability to build and traverse commit ancestry graphs. For this reason, it has a lot of different options that enable it to be used by commands as different as *git bisect* and *git repack*.

OPTIONS

1. Commit Limiting

Besides specifying a range of commits that should be listed using the special notations explained in the description, additional commit limiting may be applied.

Using more options generally further limits the output (e.g. *--since=<date1>* limits to commits newer than *<date1>*, and using it with *--grep=<pattern>* further limits to commits whose log message has a line that matches *<pattern>*), unless otherwise noted.

Note that these are applied before commit ordering and formatting options, such as *--reverse*.

-<number> , -n <number> , --max-count=<number>

Limit the number of commits to output.

--skip=<number>

Skip *number* commits before starting to show the commit output.

--since=<date> , --after=<date>

Show commits more recent than a specific date.

--since-as-filter=<date>

Show all commits more recent than a specific date. This visits all commits in the range, rather than stopping at the first commit which is older than a specific date.

--until=<date> , --before=<date>

Show commits older than a specific date.

--max-age=<timestamp> , --min-age=<timestamp>

Limit the commits output to specified time range.

--author=<pattern> , --committer=<pattern>

Limit the commits output to ones with author/committer header lines that match the specified pattern (regular expression). With more than one *--author=<pattern>*, commits whose author matches any of the given patterns are chosen (similarly for multiple *--committer=<pattern>*).

--grep-reflog=<pattern>

Limit the commits output to ones with reflog entries that match the specified pattern (regular expression). With more than one *--grep-reflog*, commits whose reflog message matches any of the given patterns are chosen. It is an error to use this option unless *--walk-reflogs* is in use.

--grep=<pattern>

Limit the commits output to ones with a log message that matches the specified pattern (regular expression). With more than one *--grep=<pattern>*, commits whose message matches any of the given patterns are chosen (but see *--all-match*).

--all-match

Limit the commits output to ones that match all given *--grep*, instead of ones that match at least one.

--invert-grep

Limit the commits output to ones with a log message that do not match the pattern specified with *--grep=<pattern>*.

-i , --regexp-ignore-case

Match the regular expression limiting patterns without regard to letter case.

--basic-regexp

Consider the limiting patterns to be basic regular expressions; this is the default.

-E , --extended-regexp

Consider the limiting patterns to be extended regular expressions instead of the default basic regular expressions.

-F , --fixed-strings

Consider the limiting patterns to be fixed strings (don't interpret pattern as a regular expression).

-P , --perl-regexp

Consider the limiting patterns to be Perl-compatible regular expressions.

Support for these types of regular expressions is an optional compile-time dependency. If Git wasn't compiled with support for them providing this option will cause it to die.

--remove-empty

Stop when a given path disappears from the tree.

--merges

Print only merge commits. This is exactly the same as *--min-parents=2*.

--no-merges

Do not print commits with more than one parent. This is exactly the same as *--max-parents=1*.

--min-parents=<number> , --max-parents=<number> , --no-min-parents , --no-max-parents

Show only commits which have at least (or at most) that many parent commits. In particular, *--max-parents=1* is the same as *--no-merges*, *--min-parents=2* is the same as *--merges*. *--max-parents=0* gives all root commits and *--min-parents=3* all octopus merges.

--no-min-parents and *--no-max-parents* reset these limits (to no limit) again. Equivalent forms are *--min-parents=0* (any commit has 0 or more parents) and *--max-parents=-1* (negative numbers denote no upper limit).

--first-parent

When finding commits to include, follow only the first parent commit upon seeing a merge commit. This option can give a better overview when viewing the evolution of a particular topic branch, because merges into a topic branch tend to be only about adjusting to updated upstream from time to time, and this option allows you to ignore the individual commits brought in to your history by such a merge.

--exclude-first-parent-only

When finding commits to exclude (with a ^), follow only the first parent commit upon seeing a merge commit. This can be used to find the set of changes in a topic branch from the point where it diverged from the remote branch, given that arbitrary merges can be valid topic branch changes.

--not

Reverses the meaning of the ^ prefix (or lack thereof) for all following revision specifiers, up to the next *--not*. When used on the command line before --stdin, the revisions passed through stdin will not be affected by it. Conversely, when passed via standard input, the revisions passed on the command line will not be affected by it.

--all

Pretend as if all the refs in *refs/*, along with *HEAD*, are listed on the command line as *<commit>*.

--branches[=<pattern>]

Pretend as if all the refs in *refs/heads* are listed on the command line as *<commit>*. If *<pattern>* is given, limit branches to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--tags[=<pattern>]

Pretend as if all the refs in *refs/tags* are listed on the command line as *<commit>*. If *<pattern>* is given, limit tags to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--remotes[=<pattern>]

Pretend as if all the refs in *refs/remotes* are listed on the command line as *<commit>*. If *<pattern>* is given, limit remote-tracking branches to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--glob=<glob-pattern>

Pretend as if all the refs matching shell glob *<glob-pattern>* are listed on the command line as *<commit>*. Leading *refs/*, is automatically prepended if missing. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--exclude=<glob-pattern>

Do not include refs matching *<glob-pattern>* that the next *--all*, *--branches*, *--tags*, *--remotes*, or *--glob* would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next *--all*, *--branches*, *--tags*, *--remotes*, or *--glob* option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with *refs/heads*, *refs/tags*, or *refs/remotes* when applied to *--branches*, *--tags*, or *--remotes*, respectively, and they must begin with *refs/* when applied to *--glob* or *--all*. If a trailing */** is intended, it must be given explicitly.

--exclude-hidden=[fetch|receive|uploadpack]

Do not include refs that would be hidden by *git-fetch*, *git-receive-pack* or *git-upload-pack* by consulting the appropriate *fetch.hideRefs*, *receive.hideRefs* or *uploadpack.hideRefs* configuration along with *transfer.hideRefs* (see Section G.3.29, “git-config(1)”). This option affects the next pseudo-ref option *--all* or *--glob* and is cleared after processing them.

--reflog

Pretend as if all objects mentioned by reflogs are listed on the command line as <*commit*>.

--alternate-refs

Pretend as if all objects mentioned as ref tips of alternate repositories were listed on the command line. An alternate repository is any repository whose object directory is specified in *objects/info/alternates*. The set of included objects may be modified by *core.alternateRefsCommand*, etc. See Section G.3.29, “git-config(1)”.

--single-worktree

By default, all working trees will be examined by the following options when there are more than one (see Section G.3.159, “git-worktree(1)”): *--all*, *--reflog* and *--indexed-objects*. This option forces them to examine the current working tree only.

--ignore-missing

Upon seeing an invalid object name in the input, pretend as if the bad input was not given.

--stdin

In addition to getting arguments from the command line, read them from standard input as well. This accepts commits and pseudo-options like *--all* and *--glob=*. When a -- separator is seen, the following input is treated as paths and used to limit the result. Flags like *--not* which are read via standard input are only respected for arguments passed in the same way and will not influence any subsequent command line arguments.

--quiet

Don't print anything to standard output. This form is primarily meant to allow the caller to test the exit status to see if a range of objects is fully connected (or not). It is faster than redirecting stdout to */dev/null* as the output does not have to be formatted.

--disk-usage , --disk-usage=human

Suppress normal output; instead, print the sum of the bytes used for on-disk storage by the selected commits or objects. This is equivalent to piping the output into *git cat-file --batch-check='%(objectsize:disk)'*, except that it runs much faster (especially with *--use-bitmap-index*). See the *CAVEATS* section in Section G.3.13, “git-cat-file(1)” for the limitations of what "on-disk storage" means. With the optional value *human*, on-disk storage size is shown in human-readable string(e.g. 12.24 Kib, 3.50 Mib).

--cherry-mark

Like *--cherry-pick* (see below) but mark equivalent commits with = rather than omitting them, and inequivalent ones with +.

--cherry-pick

Omit any commit that introduces the same change as another commit on the other side when the set of commits are limited with symmetric difference.

For example, if you have two branches, *A* and *B*, a usual way to list all commits on only one side of them is with *--left-right* (see the example below in the description of the *--left-right* option). However, it shows the commits that were cherry-picked from the other branch (for example, 3rd on b may be cherry-picked from branch A). With this option, such pairs of commits are excluded from the output.

--left-only , --right-only

List only commits on the respective side of a symmetric difference, i.e. only those which would be marked < resp. > by *--left-right*.

For example, *--cherry-pick --right-only A...B* omits those commits from *B* which are in *A* or are patch-equivalent to a commit in *A*. In other words, this lists the + commits from *git cherry A B*. More precisely, *--cherry-pick --right-only --no-merges* gives the exact list.

--cherry

A synonym for *--right-only --cherry-mark --no-merges*; useful to limit the output to the commits on our side and mark those that have been applied to the other side of a forked history with *git log --cherry upstream...mybranch*, similar to *git cherry upstream mybranch*.

-g , --walk-reflogs

Instead of walking the commit ancestry chain, walk reflog entries from the most recent one to older ones. When this option is used you cannot specify commits to exclude (that is, *^commit*, *commit1..commit2*, and *commit1...commit2* notations cannot be used).

With *--pretty* format other than *oneline* and *reference* (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. The reflog designator in the output may be shown as *ref@{<Nth>}* (where *<Nth>* is the reverse-chronological index in the reflog) or as *ref@{<timestamp>}* (with the *<timestamp>* for that entry), depending on a few rules:

1. If the starting point is specified as *ref@{<Nth>}*, show the index format.
2. If the starting point was specified as *ref@{now}*, show the timestamp format.
3. If neither was used, but *--date* was given on the command line, show the timestamp in the format requested by *--date*.
4. Otherwise, show the index format.

Under *--pretty=oneline*, the commit message is prefixed with this information on the same line. This option cannot be combined with *--reverse*. See also Section G.3.109, “git-reflog(1)”.

Under *--pretty=reference*, this information will not be shown at all.

--merge

Show commits touching conflicted paths in the range *HEAD...<other>*, where *<other>* is the first existing pseudoref in *MERGE_HEAD*, *CHERRY_PICK_HEAD*, *REVERT_HEAD* or *REBASE_HEAD*. Only works when the index has unmerged entries. This option can be used to show relevant commits when resolving conflicts from a 3-way merge.

--boundary

Output excluded boundary commits. Boundary commits are prefixed with -.

--use-bitmap-index

Try to speed up the traversal using the pack bitmap index (if one is available). Note that when traversing with *--objects*, trees and blobs will not have their associated path printed.

--progress=<header>

Show progress reports on stderr as objects are considered. The *<header>* text will be printed with each progress update.

2. History Simplification

Sometimes you are only interested in parts of the history, for example the commits modifying a particular <path>. But there are two parts of *History Simplification*, one part is selecting the commits and the other is how to do it, as there are various strategies to simplify the history.

The following options select the commits to be shown:

<paths>

Commits modifying the given <paths> are selected.

--simplify-by-decoration

Commits that are referred by some branch or tag are selected.

Note that extra commits can be shown to give a meaningful history.

The following options affect the way the simplification is performed:

Default mode

Simplifies the history to the simplest history explaining the final state of the tree. Simplest because it prunes some side branches if the end result is the same (i.e. merging branches with the same content)

--show-pulls

Include all commits from the default mode, but also any merge commits that are not TREESAME to the first parent but are TREESAME to a later parent. This mode is helpful for showing the merge commits that "first introduced" a change to a branch.

--full-history

Same as the default mode, but does not prune some history.

--dense

Only the selected commits are shown, plus some to have a meaningful history.

--sparse

All commits in the simplified history are shown.

--simplify-merges

Additional option to *--full-history* to remove some needless merges from the resulting history, as there are no selected commits contributing to this merge.

--ancestry-path[=<commit>]

When given a range of commits to display (e.g. *commit1..commit2* or *commit2 ^commit1*), only display commits in that range that are ancestors of <commit>, descendants of <commit>, or <commit> itself. If no commit is specified, use *commit1* (the excluded part of the range) as <commit>. Can be passed multiple times; if so, a commit is included if it is any of the commits given or if it is an ancestor or descendant of one of them.

A more detailed explanation follows.

Suppose you specified *foo* as the <paths>. We shall call commits that modify *foo* !TREESAME, and the rest TREESAME. (In a diff filtered for *foo*, they look different and equal, respectively.)

In the following, we will always refer to the same example history to illustrate the differences between simplification settings. We assume that you are filtering for a file *foo* in this commit graph:

```
          .-A---M---N---O---P---Q
         /     /   /   /   /   /
        I     B   C   D   E   Y
         \   /   /   /   /   /
```

```
          `-------------'   X
```

The horizontal line of history A---Q is taken to be the first parent of each merge. The commits are:

- *I* is the initial commit, in which *foo* exists with contents asdf, and a file *quux* exists with contents quux. Initial commits are compared to an empty tree, so *I* is !TREESAME.
- In *A*, *foo* contains just foo.
- *B* contains the same change as *A*. Its merge *M* is trivial and hence TREESAME to all parents.
- *C* does not change *foo*, but its merge *N* changes it to foobar, so it is not TREESAME to any parent.
- *D* sets *foo* to baz. Its merge *O* combines the strings from *N* and *D* to foobarbaz; i.e., it is not TREESAME to any parent.
- *E* changes *quux* to xyzzy, and its merge *P* combines the strings to quux xyzzy. *P* is TREESAME to *O*, but not to *E*.
- *X* is an independent root commit that added a new file *side*, and *Y* modified it. *Y* is TREESAME to *X*. Its merge *Q* added *side* to *P*, and *Q* is TREESAME to *P*, but not to *Y*.

rev-list walks backwards through history, including or excluding commits based on whether *--full-history* and/or parent rewriting (via *--parents* or *--children*) are used. The following settings are available.

Default mode

Commits are included if they are not TREESAME to any parent (though this can be changed, see *--sparse* below). If the commit was a merge, and it was TREESAME to one parent, follow only that parent. (Even if there are several TREESAME parents, follow only one of them.) Otherwise, follow all parents.

This results in:

```
     .-A---N---O
    /     /   /
   I---------D
```

Note how the rule to only follow the TREESAME parent, if one is available, removed *B* from consideration entirely. *C* was considered via *N*, but is TREESAME. Root commits are compared to an empty tree, so *I* is !TREESAME.

Parent/child relations are only visible with *--parents*, but that does not affect the commits selected in default mode, so we have shown the parent lines.

--full-history without parent rewriting

This mode differs from the default in one point: always follow all parents of a merge, even if it is TREESAME to one of them. Even if more than one side of the merge has commits that are included, this does not imply that the merge itself is! In the example, we get

```
I  A  B  N  D  O  P  Q
```

M was excluded because it is TREESAME to both parents. *E*, *C* and *B* were all walked, but only *B* was !TREESAME, so the others do not appear.

Note that without parent rewriting, it is not really possible to talk about the parent/child relationships between the commits, so we show them disconnected.

--full-history with parent rewriting

Ordinary commits are only included if they are !TREESAME (though this can be changed, see *--sparse* below).

Merges are always included. However, their parent list is rewritten: Along each parent, prune away commits that are not included themselves. This results in

```
  .-A---M---N---O---P---Q
 /     /   /   /   /
I     B   /   D   /
 \   /   /   /   /
  `-------------'
```

Compare to *--full-history* without rewriting above. Note that *E* was pruned away because it is TREESAME, but the parent list of P was rewritten to contain *E*'s parent *I*. The same happened for *C* and *N*, and *X*, *Y* and *Q*.

In addition to the above settings, you can change whether TREESAME affects inclusion:

--dense

Commits that are walked are included if they are not TREESAME to any parent.

--sparse

All commits that are walked are included.

Note that without *--full-history*, this still simplifies merges: if one of the parents is TREESAME, we follow only that one, so the other sides of the merge are never walked.

--simplify-merges

First, build a history graph in the same way that *--full-history* with parent rewriting does (see above).

Then simplify each commit *C* to its replacement *C'* in the final history according to the following rules:

- Set *C'* to *C*.
- Replace each parent *P* of *C'* with its simplification *P'*. In the process, drop parents that are ancestors of other parents or that are root commits TREESAME to an empty tree, and remove duplicates, but take care to never drop all parents that we are TREESAME to.
- If after this parent rewriting, *C'* is a root or merge commit (has zero or >1 parents), a boundary commit, or !TREESAME, it remains. Otherwise, it is replaced with its only parent.

The effect of this is best shown by way of comparing to *--full-history* with parent rewriting. The example turns into:

```
  .-A---M---N---O
 /     /       /
I     B       D
 \   /       /
  `---------'
```

Note the major differences in *N*, *P*, and *Q* over *--full-history*:

- *N*'s parent list had *I* removed, because it is an ancestor of the other parent *M*. Still, *N* remained because it is !TREESAME.
- *P*'s parent list similarly had *I* removed. *P* was then removed completely, because it had one parent and is TREESAME.
- *Q*'s parent list had *Y* simplified to *X*. *X* was then removed, because it was a TREESAME root. *Q* was then removed completely, because it had one parent and is TREESAME.

There is another simplification mode available:

--ancestry-path[=<commit>]

Limit the displayed commits to those which are an ancestor of <commit>, or which are a descendant of <commit>, or are <commit> itself.

As an example use case, consider the following commit history:

```
          D---E-------F
         /     \       \
        B---C---G---H---I---J
       /                     \
      A-------K---------------L--M
```

A regular *D..M* computes the set of commits that are ancestors of *M*, but excludes the ones that are ancestors of *D*. This is useful to see what happened to the history leading to *M* since *D*, in the sense that what does *M* have that did not exist in *D*. The result in this example would be all the commits, except *A* and *B* (and *D* itself, of course).

When we want to find out what commits in *M* are contaminated with the bug introduced by *D* and need fixing, however, we might want to view only the subset of *D..M* that are actually descendants of *D*, i.e. excluding *C* and *K*. This is exactly what the *--ancestry-path* option does. Applied to the *D..M* range, it results in:

```
E-------F
 \       \
  G---H---I---J
                \
                 L--M
```

We can also use *--ancestry-path=D* instead of *--ancestry-path* which means the same thing when applied to the *D..M* range but is just more explicit.

If we instead are interested in a given topic within this range, and all commits affected by that topic, we may only want to view the subset of *D..M* which contain that topic in their ancestry path. So, using *--ancestry-path=H D..M* for example would result in:

```
E
 \
  G---H---I---J
                \
                 L--M
```

Whereas *--ancestry-path=K D..M* would result in

```
K---------------L--M
```

Before discussing another option, *--show-pulls*, we need to create a new example history.

A common problem users face when looking at simplified history is that a commit they know changed a file somehow does not appear in the file's simplified history. Let's demonstrate a new example and show how options such as *--full-history* and *--simplify-merges* works in that case:

```
     .-A---M-----C--N---O---P
    /     / \  \  \/   /   /
   I     B   \  R-'`-Z'   /
    \   /     \/         /
     \ /      /\        /
      `---X--'   `---Y--'
```

For this example, suppose *I* created *file.txt* which was modified by *A*, *B*, and *X* in different ways. The single-parent commits *C*, *Z*, and *Y* do not change *file.txt*. The merge commit *M* was created by resolving the merge conflict to include both changes from *A* and *B* and hence is not TREESAME to either. The merge commit *R*, however, was

created by ignoring the contents of *file.txt* at *M* and taking only the contents of *file.txt* at *X*. Hence, *R* is TREESAME to *X* but not *M*. Finally, the natural merge resolution to create *N* is to take the contents of *file.txt* at *R*, so *N* is TREESAME to *R* but not *C*. The merge commits *O* and *P* are TREESAME to their first parents, but not to their second parents, *Z* and *Y* respectively.

When using the default mode, *N* and *R* both have a TREESAME parent, so those edges are walked and the others are ignored. The resulting history graph is:

```
I---X
```

When using *--full-history*, Git walks every edge. This will discover the commits *A* and *B* and the merge *M*, but also will reveal the merge commits *O* and *P*. With parent rewriting, the resulting graph is:

```
  .-A---M--------N---O---P
 /     / \  \  \/   /   /
I     B   \  R-'`--'   /
 \   /     \/         /
  \ /      /\        /
   `---X--'  `------'
```

Here, the merge commits *O* and *P* contribute extra noise, as they did not actually contribute a change to *file.txt*. They only merged a topic that was based on an older version of *file.txt*. This is a common issue in repositories using a workflow where many contributors work in parallel and merge their topic branches along a single trunk: many unrelated merges appear in the *--full-history* results.

When using the *--simplify-merges* option, the commits *O* and *P* disappear from the results. This is because the rewritten second parents of *O* and *P* are reachable from their first parents. Those edges are removed and then the commits look like single-parent commits that are TREESAME to their parent. This also happens to the commit *N*, resulting in a history view as follows:

```
  .-A---M--.
 /     /    \
I     B      R
 \   /      /
  \ /      /
   `---X--'
```

In this view, we see all of the important single-parent changes from *A*, *B*, and *X*. We also see the carefully-resolved merge *M* and the not-so-carefully-resolved merge *R*. This is usually enough information to determine why the commits *A* and *B* "disappeared" from history in the default view. However, there are a few issues with this approach.

The first issue is performance. Unlike any previous option, the *--simplify-merges* option requires walking the entire commit history before returning a single result. This can make the option difficult to use for very large repositories.

The second issue is one of auditing. When many contributors are working on the same repository, it is important which merge commits introduced a change into an important branch. The problematic merge *R* above is not likely to be the merge commit that was used to merge into an important branch. Instead, the merge *N* was used to merge *R* and *X* into the important branch. This commit may have information about why the change *X* came to override the changes from *A* and *B* in its commit message.

--show-pulls

In addition to the commits shown in the default history, show each merge commit that is not TREESAME to its first parent but is TREESAME to a later parent.

When a merge commit is included by *--show-pulls*, the merge is treated as if it "pulled" the change from another branch. When using *--show-pulls* on this example (and no other options) the resulting graph is:

```
I---X---R---N
```

Here, the merge commits *R* and *N* are included because they pulled the commits *X* and *R* into the base branch, respectively. These merges are the reason the commits *A* and *B* do not appear in the default history.

When *--show-pulls* is paired with *--simplify-merges*, the graph includes all of the necessary information:

```
   .-A---M--.   N
  /     /    \ /
 I     B      R
  \   /      /
   \ /      /
    `---X--'
```

Notice that since *M* is reachable from *R*, the edge from *N* to *M* was simplified away. However, *N* still appears in the history as an important commit because it "pulled" the change *R* into the main branch.

The *--simplify-by-decoration* option allows you to view only the big picture of the topology of the history, by omitting commits that are not referenced by tags. Commits are marked as !TREESAME (in other words, kept after history simplification rules described above) if (1) they are referenced by tags, or (2) they change the contents of the paths given on the command line. All other commits are marked as TREESAME (subject to be simplified away).

3. Bisection Helpers

--bisect

Limit output to the one commit object which is roughly halfway between included and excluded commits. Note that the bad bisection ref *refs/bisect/bad* is added to the included commits (if it exists) and the good bisection refs *refs/bisect/good-** are added to the excluded commits (if they exist). Thus, supposing there are no refs in *refs/bisect/*, if

```
$ git rev-list --bisect foo ^bar ^baz
```

outputs *midpoint*, the output of the two commands

```
$ git rev-list foo ^midpoint
$ git rev-list midpoint ^bar ^baz
```

would be of roughly the same length. Finding the change which introduces a regression is thus reduced to a binary search: repeatedly generate and test new 'midpoint's until the commit chain is of length one.

--bisect-vars

This calculates the same as *--bisect*, except that refs in *refs/bisect/* are not used, and except that this outputs text ready to be eval'ed by the shell. These lines will assign the name of the midpoint revision to the variable *bisect_rev*, and the expected number of commits to be tested after *bisect_rev* is tested to *bisect_nr*, the expected number of commits to be tested if *bisect_rev* turns out to be good to *bisect_good*, the expected number of commits to be tested if *bisect_rev* turns out to be bad to *bisect_bad*, and the number of commits we are bisecting right now to *bisect_all*.

--bisect-all

This outputs all the commit objects between the included and excluded commits, ordered by their distance to the included and excluded commits. Refs in *refs/bisect/* are not used. The farthest from them is displayed first. (This is the only one displayed by *--bisect*.)

This is useful because it makes it easy to choose a good commit to test when you want to avoid to test some of them for some reason (they may not compile for example).

This option can be used along with *--bisect-vars*, in this case, after all the sorted commit objects, there will be the same text as if *--bisect-vars* had been used alone.

4. Commit Ordering

By default, the commits are shown in reverse chronological order.

--date-order

Show no parents before all of its children are shown, but otherwise show commits in the commit timestamp order.

--author-date-order

Show no parents before all of its children are shown, but otherwise show commits in the author timestamp order.

--topo-order

Show no parents before all of its children are shown, and avoid showing commits on multiple lines of history intermixed.

For example, in a commit history like this:

```
---1----2----4----7
    \               \
     3----5----6----8---
```

where the numbers denote the order of commit timestamps, *git rev-list* and friends with *--date-order* show the commits in the timestamp order: 8 7 6 5 4 3 2 1.

With *--topo-order*, they would show 8 6 5 3 7 4 2 1 (or 8 7 4 2 6 5 3 1); some older commits are shown before newer ones in order to avoid showing the commits from two parallel development track mixed together.

--reverse

Output the commits chosen to be shown (see Commit Limiting section above) in reverse order. Cannot be combined with *--walk-reflogs*.

5. Object Traversal

These options are mostly targeted for packing of Git repositories.

--objects

Print the object IDs of any object referenced by the listed commits. *--objects foo ^bar* thus means send me all object IDs which I need to download if I have the commit object *bar* but not *foo*. See also *--object-names* below.

--in-commit-order

Print tree and blob ids in order of the commits. The tree and blob ids are printed after they are first referenced by a commit.

--objects-edge

Similar to *--objects*, but also print the IDs of excluded commits prefixed with a - character. This is used by Section G.3.96, “git-pack-objects(1)” to build a thin pack, which records objects in deltified form based on objects contained in these excluded commits to reduce network traffic.

--objects-edge-aggressive

Similar to *--objects-edge*, but it tries harder to find excluded commits at the cost of increased time. This is used instead of *--objects-edge* to build thin packs for shallow repositories.

--indexed-objects

Pretend as if all trees and blobs used by the index are listed on the command line. Note that you probably want to use *--objects*, too.

--unpacked

Only useful with *--objects*; print the object IDs that are not in packs.

--object-names

Only useful with *--objects*; print the names of the object IDs that are found. This is the default behavior. Note that the "name" of each object is ambiguous, and mostly intended as a hint for packing objects. In particular: no distinction is made between the names of tags, trees, and blobs; path names may be modified to remove newlines; and if an object would appear multiple times with different names, only one name is shown.

--no-object-names

Only useful with *--objects*; does not print the names of the object IDs that are found. This inverts *--object-names*. This flag allows the output to be more easily parsed by commands such as Section G.3.13, "git-cat-file(1)".

--filter=<filter-spec>

Only useful with one of the *--objects**; omits objects (usually blobs) from the list of printed objects. The *<filter-spec>* may be one of the following:

The form *--filter=blob:none* omits all blobs.

The form *--filter=blob:limit=<n>[kmg]* omits blobs of size at least n bytes or units. n may be zero. The suffixes k, m, and g can be used to name units in KiB, MiB, or GiB. For example, *blob:limit=1k* is the same as *blob:limit=1024*.

The form *--filter=object:type=(tag|commit|tree|blob)* omits all objects which are not of the requested type.

The form *--filter=sparse:oid=<blob-ish>* uses a sparse-checkout specification contained in the blob (or blob-expression) *<blob-ish>* to omit blobs that would not be required for a sparse checkout on the requested refs.

The form *--filter=tree:<depth>* omits all blobs and trees whose depth from the root tree is >= <depth> (minimum depth if an object is located at multiple depths in the commits traversed). <depth>=0 will not include any trees or blobs unless included explicitly in the command-line (or standard input when --stdin is used). <depth>=1 will include only the tree and blobs which are referenced directly by a commit reachable from <commit> or an explicitly-given object. <depth>=2 is like <depth>=1 while also including trees and blobs one more level removed from an explicitly-given commit or tree.

Note that the form *--filter=sparse:path=<path>* that wants to read from an arbitrary path on the filesystem has been dropped for security reasons.

Multiple *--filter=* flags can be specified to combine filters. Only objects which are accepted by every filter are included.

The form *--filter=combine:<filter1>+<filter2>+…<filterN>* can also be used to combined several filters, but this is harder than just repeating the *--filter* flag and is usually not necessary. Filters are joined by + and individual filters are %-encoded (i.e. URL-encoded). Besides the + and *%* characters, the following characters are reserved and also must be encoded: *~!@#$^&*()[]{}\;",<>?'`* as well as all characters with ASCII code *<= 0x20*, which includes space and newline.

Other arbitrary characters can also be encoded. For instance, *combine:tree:3+blob:none* and *combine:tree%3A3+blob%3Anone* are equivalent.

--no-filter

Turn off any previous *--filter=* argument.

--filter-provided-objects

Filter the list of explicitly provided objects, which would otherwise always be printed even if they did not match any of the filters. Only useful with *--filter=*.

--filter-print-omitted

Only useful with *--filter=*; prints a list of the objects omitted by the filter. Object IDs are prefixed with a ~ character.

--missing=<missing-action>

A debug option to help with future "partial clone" development. This option specifies how missing objects are handled.

The form *--missing=error* requests that rev-list stop with an error if a missing object is encountered. This is the default action.

The form *--missing=allow-any* will allow object traversal to continue if a missing object is encountered. Missing objects will silently be omitted from the results.

The form *--missing=allow-promisor* is like *allow-any*, but will only allow object traversal to continue for EXPECTED promisor missing objects. Unexpected missing objects will raise an error.

The form *--missing=print* is like *allow-any*, but will also print a list of the missing objects. Object IDs are prefixed with a ? character.

If some tips passed to the traversal are missing, they will be considered as missing too, and the traversal will ignore them. In case we cannot get their Object ID though, an error will be raised.

--exclude-promisor-objects

(For internal use only.) Prefilter object traversal at promisor boundary. This is used with partial clone. This is stronger than *--missing=allow-promisor* because it limits the traversal, rather than just silencing errors about missing objects.

--no-walk[=(sorted|unsorted)]

Only show the given commits, but do not traverse their ancestors. This has no effect if a range is specified. If the argument *unsorted* is given, the commits are shown in the order they were given on the command line. Otherwise (if *sorted* or no argument was given), the commits are shown in reverse chronological order by commit time. Cannot be combined with *--graph*.

--do-walk

Overrides a previous *--no-walk*.

6. Commit Formatting

Using these options, Section G.3.120, “git-rev-list(1)” will act similar to the more specialized family of commit log tools: Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, and Section G.3.158, “git-whatchanged(1)”

--pretty[=<format>] , --format=<format>

Pretty-print the contents of the commit logs in a given format, where *<format>* can be one of *oneline*, *short*, *medium*, *full*, *fuller*, *reference*, *email*, *raw*, *format:<string>* and *tformat:<string>*. When *<format>* is none of the above, and has *%placeholder* in it, it acts as if *--pretty=tformat:<format>* were given.

See the "PRETTY FORMATS" section for some additional details for each format. When *=<format>* part is omitted, it defaults to *medium*.

Note: you can specify the default pretty format in the repository configuration (see Section G.3.29, “git-config(1)”).

--abbrev-commit

Instead of showing the full 40-byte hexadecimal commit object name, show a prefix that names the object uniquely. "--abbrev=<n>" (which also modifies diff output, if it is displayed) option can be used to specify the minimum length of the prefix.

This should make "--pretty=oneline" a whole lot more readable for people using 80-column terminals.

--no-abbrev-commit

Show the full 40-byte hexadecimal commit object name. This negates *--abbrev-commit*, either explicit or implied by other options such as "--oneline". It also overrides the *log.abbrevCommit* variable.

--oneline

This is a shorthand for "--pretty=oneline --abbrev-commit" used together.

--encoding=<encoding>

Commit objects record the character encoding used for the log message in their encoding header; this option can be used to tell the command to re-code the commit log message in the encoding preferred by the user. For non plumbing commands this defaults to UTF-8. Note that if an object claims to be encoded in *X* and we are outputting in *X*, we will output the object verbatim; this means that invalid sequences in the original commit may be copied to the output. Likewise, if iconv(3) fails to convert the commit, we will quietly output the original object verbatim.

--expand-tabs=<n> , --expand-tabs , --no-expand-tabs

Perform a tab expansion (replace each tab with enough spaces to fill to the next display column that is a multiple of *<n>*) in the log message before showing it in the output. *--expand-tabs* is a short-hand for *--expand-tabs=8*, and *--no-expand-tabs* is a short-hand for *--expand-tabs=0*, which disables tab expansion.

By default, tabs are expanded in pretty formats that indent the log message by 4 spaces (i.e. *medium*, which is the default, *full*, and *fuller*).

--show-signature

Check the validity of a signed commit object by passing the signature to *gpg --verify* and show the output.

--relative-date

Synonym for *--date=relative*.

--date=<format>

Only takes effect for dates shown in human-readable format, such as when using *--pretty*. *log.date* config variable sets a default value for the log command's *--date* option. By default, dates are shown in the original time zone (either committer's or author's). If *-local* is appended to the format (e.g., *iso-local*), the user's local time zone is used instead.

--date=relative shows dates relative to the current time, e.g. 2 hours ago. The *-local* option has no effect for *--date=relative*.

--date=local is an alias for *--date=default-local*.

--date=iso (or *--date=iso8601*) shows timestamps in a ISO 8601-like format. The differences to the strict ISO 8601 format are:

- a space instead of the *T* date/time delimiter
- a space between time and time zone

- no colon between hours and minutes of the time zone

--date=iso-strict (or *--date=iso8601-strict*) shows timestamps in strict ISO 8601 format.

--date=rfc (or *--date=rfc2822*) shows timestamps in RFC 2822 format, often found in email messages.

--date=short shows only the date, but not the time, in *YYYY-MM-DD* format.

--date=raw shows the date as seconds since the epoch (1970-01-01 00:00:00 UTC), followed by a space, and then the timezone as an offset from UTC (a + or - with four digits; the first two are hours, and the second two are minutes). I.e., as if the timestamp were formatted with *strftime("%s %z")*). Note that the *-local* option does not affect the seconds-since-epoch value (which is always measured in UTC), but does switch the accompanying timezone value.

--date=human shows the timezone if the timezone does not match the current time-zone, and doesn't print the whole date if that matches (ie skip printing year for dates that are "this year", but also skip the whole date itself if it's in the last few days and we can just say what weekday it was). For older dates the hour and minute is also omitted.

--date=unix shows the date as a Unix epoch timestamp (seconds since 1970). As with *--raw*, this is always in UTC and therefore *-local* has no effect.

--date=format:... feeds the format ... to your system *strftime*, except for %s, %z, and %Z, which are handled internally. Use *--date=format:%c* to show the date in your system locale's preferred format. See the *strftime* manual for a complete list of format placeholders. When using *-local*, the correct syntax is *--date=format-local:...*.

--date=default is the default format, and is based on ctime(3) output. It shows a single line with three-letter day of the week, three-letter month, day-of-month, hour-minute-seconds in "HH:MM:SS" format, followed by 4-digit year, plus timezone information, unless the local time zone is used, e.g. *Thu Jan 1 00:00:00 1970 +0000*.

--header

Print the contents of the commit in raw-format; each record is separated with a NUL character.

--no-commit-header

Suppress the header line containing "commit" and the object ID printed before the specified format. This has no effect on the built-in formats; only custom formats are affected.

--commit-header

Overrides a previous *--no-commit-header*.

--parents

Print also the parents of the commit (in the form "commit parent…"). Also enables parent rewriting, see *History Simplification* above.

--children

Print also the children of the commit (in the form "commit child…"). Also enables parent rewriting, see *History Simplification* above.

--timestamp

Print the raw commit timestamp.

--left-right

Mark which side of a symmetric difference a commit is reachable from. Commits from the left side are prefixed with < and those from the right with >. If combined with *--boundary*, those commits are prefixed with -.

For example, if you have this topology:

```
              y---b---b  branch B
             / \ /
            /   .
           /   / \
          o---x---a---a  branch A
```

you would get an output like this:

```
$ git rev-list --left-right --boundary --pretty=oneline A...B

>bbbbbbb... 3rd on b
>bbbbbbb... 2nd on b
<aaaaaaa... 3rd on a
<aaaaaaa... 2nd on a
-yyyyyyy... 1st on b
-xxxxxxx... 1st on a
```

--graph

Draw a text-based graphical representation of the commit history on the left hand side of the output. This may cause extra lines to be printed in between commits, in order for the graph history to be drawn properly. Cannot be combined with *--no-walk*.

This enables parent rewriting, see *History Simplification* above.

This implies the *--topo-order* option by default, but the *--date-order* option may also be specified.

--show-linear-break[=<barrier>]

When --graph is not used, all history branches are flattened which can make it hard to see that the two consecutive commits do not belong to a linear branch. This option puts a barrier in between them in that case. If *<barrier>* is specified, it is the string that will be shown instead of the default one.

--count

Print a number stating how many commits would have been listed, and suppress all other output. When used together with *--left-right*, instead print the counts for left and right commits, separated by a tab. When used together with *--cherry-mark*, omit patch equivalent commits from these counts and print the count for equivalent commits separated by a tab.

PRETTY FORMATS

If the commit is a merge, and if the pretty-format is not *oneline*, *email* or *raw*, an additional line is inserted before the *Author:* line. This line begins with "Merge: " and the hashes of ancestral commits are printed, separated by spaces. Note that the listed commits may not necessarily be the list of the **direct** parent commits if you have limited your view of history: for example, if you are only interested in changes related to a certain directory or file.

There are several built-in formats, and you can define additional formats by setting a pretty.<name> config option to either another format name, or a *format:* string, as described below (see Section G.3.29, “git-config(1)”). Here are the details of the built-in formats:

- *oneline*

  ```
  <hash> <title-line>
  ```

  This is designed to be as compact as possible.

- *short*

```
commit <hash>
Author: <author>

<title-line>
```

- *medium*

```
commit <hash>
Author: <author>
Date:   <author-date>

<title-line>

<full-commit-message>
```

- *full*

```
commit <hash>
Author: <author>
Commit: <committer>

<title-line>

<full-commit-message>
```

- *fuller*

```
commit <hash>
Author:     <author>
AuthorDate: <author-date>
Commit:     <committer>
CommitDate: <committer-date>

<title-line>

<full-commit-message>
```

- *reference*

```
<abbrev-hash> (<title-line>, <short-author-date>)
```

This format is used to refer to another commit in a commit message and is the same as *--pretty='format:%C(auto)%h (%s, %ad)'*. By default, the date is formatted with *--date=short* unless another *--date* option is explicitly specified. As with any *format:* with format placeholders, its output is not affected by other options like *--decorate* and *--walk-reflogs*.

- *email*

```
From <hash> <date>
From: <author>
Date: <author-date>
Subject: [PATCH] <title-line>

<full-commit-message>
```

- *mboxrd*

Like *email*, but lines in the commit message starting with "From " (preceded by zero or more ">") are quoted with ">" so they aren't confused as starting a new commit.

- *raw*

The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the hashes are displayed in full, regardless of whether --abbrev or --no-abbrev are used, and *parents* information show the true parent commits, without taking grafts or history simplification into account. Note that this format affects the way commits are displayed, but not the way the diff is shown e.g. with *git log --raw*. To get full object names in a raw diff format, use *--no-abbrev*.

- *format:<format-string>*

  The *format:<format-string>* format allows you to specify which information you want to show. It works a little bit like printf format, with the notable exception that you get a newline with *%n* instead of *\n*.

  E.g, *format:"The author of %h was %an, %ar%nThe title was >>%s<<%n"* would show something like this:

```
The author of fe6e0ee was Junio C Hamano, 23 hours ago
The title was >>t4119: test autocomputing -p<n> for traditional diff
 input.<<
```

  The placeholders are:

  - Placeholders that expand to a single literal character:

    %n

    newline

    %%

    a raw *%*

    %x00

    %x followed by two hexadecimal digits is replaced with a byte with the hexadecimal digits' value (we will call this "literal formatting code" in the rest of this document).

  - Placeholders that affect formatting of later placeholders:

    %Cred

    switch color to red

    %Cgreen

    switch color to green

    %Cblue

    switch color to blue

    %Creset

    reset color

    %C(…)

    color specification, as described under Values in the "CONFIGURATION FILE" section of Section G.3.29, “git-config(1)”. By default, colors are shown only when enabled for log output (by *color.diff*, *color.ui*, or *--color*, and respecting the *auto* settings of the former if we are going to a terminal). *%C(auto,...)* is accepted as a historical synonym for the default (e.g., *%C(auto,red)*). Specifying *%C(always,...)* will show the colors even when color is not otherwise enabled (though consider just using *--color=always* to enable color for the whole output, including this format and anything else git might color). *auto* alone (i.e. *%C(auto)*) will turn on auto coloring on the next placeholders until the color is switched again.

%m

left (<), right (>) or boundary (-) mark

%w([<w>[,<i1>[,<i2>]]])

switch line wrapping, like the -w option of Section G.3.130, “git-shortlog(1)”.

%<(<N> [,trunc|ltrunc|mtrunc])

make the next placeholder take at least N column widths, padding spaces on the right if necessary. Optionally truncate (with ellipsis ..) at the left (ltrunc) *..ft*, the middle (mtrunc) *mi..le*, or the end (trunc) *rig..*, if the output is longer than N columns. Note 1: that truncating only works correctly with N >= 2. Note 2: spaces around the N and M (see below) values are optional. Note 3: Emojis and other wide characters will take two display columns, which may over-run column boundaries. Note 4: decomposed character combining marks may be misplaced at padding boundaries.

%<|(<M>)

make the next placeholder take at least until Mth display column, padding spaces on the right if necessary. Use negative M values for column positions measured from the right hand edge of the terminal window.

%>(<N>), %>|(<M>)

similar to *%<(<N>), %<|(<M>)* respectively, but padding spaces on the left

%>>(<N>), %>>|(<M>)

similar to *%>(<N>), %>|(<M>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces

%><(<N>), %><|(<M>)

similar to *%<(<N>), %<|(<M>)* respectively, but padding both sides (i.e. the text is centered)

- Placeholders that expand to information extracted from the commit:

%H

commit hash

%h

abbreviated commit hash

%T

tree hash

%t

abbreviated tree hash

%P

parent hashes

%p

abbreviated parent hashes

%an

author name

%aN

author name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ae

author email

%aE

author email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%al

author email local-part (the part before the @ sign)

%aL

author local-part (see *%al*) respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ad

author date (format respects --date= option)

%aD

author date, RFC2822 style

%ar

author date, relative

%at

author date, UNIX timestamp

%ai

author date, ISO 8601-like format

%aI

author date, strict ISO 8601 format

%as

author date, short format (*YYYY-MM-DD*)

%ah

author date, human style (like the *--date=human* option of Section G.3.120, “git-rev-list(1)”)

%cn

committer name

%cN

committer name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ce

committer email

%cE

committer email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%cl

committer email local-part (the part before the @ sign)

%cL

committer local-part (see *%cl*) respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%cd

committer date (format respects --date= option)

%cD

committer date, RFC2822 style

%cr

committer date, relative

%ct

committer date, UNIX timestamp

%ci

committer date, ISO 8601-like format

%cI

committer date, strict ISO 8601 format

%cs

committer date, short format (*YYYY-MM-DD*)

%ch

committer date, human style (like the *--date=human* option of Section G.3.120, “git-rev-list(1)”)

%d

ref names, like the --decorate option of Section G.3.74, “git-log(1)”

%D

ref names without the " (", ")" wrapping.

%(decorate[:<options>])

ref names with custom decorations. The *decorate* string may be followed by a colon and zero or more comma-separated options. Option values may contain literal formatting codes. These must be used for commas (*%x2C*) and closing parentheses (*%x29*), due to their role in the option syntax.

- *prefix=<value>*: Shown before the list of ref names. Defaults to " *(*".
- *suffix=<value>*: Shown after the list of ref names. Defaults to "*)*".
- *separator=<value>*: Shown between ref names. Defaults to ", ".
- *pointer=<value>*: Shown between HEAD and the branch it points to, if any. Defaults to " -> ".
- *tag=<value>*: Shown before tag names. Defaults to "*tag:* ".

For example, to produce decorations with no wrapping or tag annotations, and spaces as separators:

%(decorate:prefix=,suffix=,tag=,separator=)

%(describe[:<options>])

human-readable name, like Section G.3.39, “git-describe(1)”; empty string for undescribable commits. The *describe* string may be followed by a colon and zero or more comma-separated options. Descriptions can be inconsistent when tags are added or removed at the same time.

- *tags[=<bool-value>]*: Instead of only considering annotated tags, consider lightweight tags as well.
- *abbrev=<number>*: Instead of using the default number of hexadecimal digits (which will vary according to the number of objects in the repository with a default of 7) of the abbreviated object name, use <number> digits, or as many digits as needed to form a unique object name.
- *match=<pattern>*: Only consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix.
- *exclude=<pattern>*: Do not consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix.

%S

ref name given on the command line by which the commit was reached (like *git log --source*), only works with *git log*

%e

encoding

%s

subject

%f

sanitized subject line, suitable for a filename

%b

body

%B

raw body (unwrapped subject and body)

%GG

raw verification message from GPG for a signed commit

%G?

show "G" for a good (valid) signature, "B" for a bad signature, "U" for a good signature with unknown validity, "X" for a good signature that has expired, "Y" for a good signature made by an expired key, "R" for a good signature made by a revoked key, "E" if the signature cannot be checked (e.g. missing key) and "N" for no signature

%GS

show the name of the signer for a signed commit

%GK

show the key used to sign a signed commit

%GF

show the fingerprint of the key used to sign a signed commit

%GP

show the fingerprint of the primary key whose subkey was used to sign a signed commit

%GT

show the trust level for the key used to sign a signed commit

%gD

reflog selector, e.g., *refs/stash@{1}* or *refs/stash@{2 minutes ago}*; the format follows the rules described for the *-g* option. The portion before the @ is the refname as given on the command line (so *git log -g refs/heads/master* would yield *refs/heads/master@{0}*).

%gd

shortened reflog selector; same as *%gD*, but the refname portion is shortened for human readability (so *refs/heads/master* becomes just *master*).

%gn

reflog identity name

%gN

reflog identity name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ge

reflog identity email

%gE

reflog identity email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%gs

reflog subject

%(trailers[:<options>])

display the trailers of the body as interpreted by Section G.3.73, “git-interpret-trailers(1)”. The *trailers* string may be followed by a colon and zero or more comma-separated options. If any option is provided multiple times, the last occurrence wins.

- *key=<key>*: only show trailers with specified <key>. Matching is done case-insensitively and trailing colon is optional. If option is given multiple times trailer lines matching any of the keys are shown. This option automatically enables the *only* option so that non-trailer lines in the trailer block are hidden. If that is not desired it can be disabled with *only=false*. E.g., *%(trailers:key=Reviewed-by)* shows trailer lines with key *Reviewed-by*.
- *only[=<bool>]*: select whether non-trailer lines from the trailer block should be included.
- *separator=<sep>*: specify the separator inserted between trailer lines. Defaults to a line feed character. The string <sep> may contain the literal formatting codes described above. To use comma as separator one must use *%x2C* as it would otherwise be parsed as next option. E.g., *%(trailers:key=Ticket,separator=%x2C)* shows all trailer lines whose key is "Ticket" separated by a comma and a space.
- *unfold[=<bool>]*: make it behave as if interpret-trailer's *--unfold* option was given. E.g., *%(trailers:only,unfold=true)* unfolds and shows all trailer lines.
- *keyonly[=<bool>]*: only show the key part of the trailer.
- *valueonly[=<bool>]*: only show the value part of the trailer.
- *key_value_separator=<sep>*: specify the separator inserted between the key and value of each trailer. Defaults to ": ". Otherwise it shares the same semantics as *separator=<sep>* above.

> **Note**
>
> Some placeholders may depend on other options given to the revision traversal engine. For example, the *%g** reflog options will insert an empty string unless we are traversing reflog entries (e.g., by *git log -g*). The *%d* and *%D* placeholders will use the "short" decoration format if *--decorate* was not already provided on the command line.

The boolean options accept an optional value *[=<bool-value>]*. The values *true*, *false*, *on*, *off* etc. are all accepted. See the "boolean" sub-section in "EXAMPLES" in Section G.3.29, “git-config(1)”. If a boolean option is given with no value, it's enabled.

If you add a + (plus sign) after % of a placeholder, a line-feed is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

If you add a - (minus sign) after % of a placeholder, all consecutive line-feeds immediately preceding the expansion are deleted if and only if the placeholder expands to an empty string.

If you add a ` ` (space) after % of a placeholder, a space is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

- *tformat:*

The *tformat:* format works exactly like *format:*, except that it provides "terminator" semantics instead of "separator" semantics. In other words, each commit has the message terminator character (usually a newline) appended, rather than a separator placed between entries. This means that the final entry of a single-line format will be properly terminated with a new line, just as the "oneline" format does. For example:

```
$ git log -2 --pretty=format:%h 4da45bef \
  | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
4da45be
7134973 -- NO NEWLINE

$ git log -2 --pretty=tformat:%h 4da45bef \
  | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
4da45be
7134973
```

In addition, any unrecognized string that has a *%* in it is interpreted as if it has *tformat:* in front of it. For example, these two are equivalent:

```
$ git log -2 --pretty=tformat:%h 4da45bef
$ git log -2 --pretty=%h 4da45bef
```

EXAMPLES

- Print the list of commits reachable from the current branch.

  ```
  git rev-list HEAD
  ```

- Print the list of commits on this branch, but not present in the upstream branch.

  ```
  git rev-list @{upstream}..HEAD
  ```

- Format commits with their author and commit message (see also the porcelain Section G.3.74, “git-log(1)”).

  ```
  git rev-list --format=medium HEAD
  ```

- Format commits along with their diffs (see also the porcelain Section G.3.74, “git-log(1)”, which can do this in a single process).

  ```
  git rev-list HEAD |
  git diff-tree --stdin --format=medium -p
  ```

- Print the list of commits on the current branch that touched any file in the *Documentation* directory.

  ```
  git rev-list HEAD -- Documentation/
  ```

- Print the list of commits authored by you in the past year, on any branch, tag, or other ref.

  ```
  git rev-list --author=you@example.com --since=1.year.ago --all
  ```

- Print the list of objects reachable from the current branch (i.e., all commits and the blobs and trees they contain).

  ```
  git rev-list --objects HEAD
  ```

- Compare the disk size of all reachable objects, versus those reachable from reflogs, versus the total packed size. This can tell you whether running *git repack -ad* might reduce the repository size (by dropping unreachable objects), and whether expiring reflogs might help.

  ```
  # reachable objects
  git rev-list --disk-usage --objects --all
  # plus reflogs
  ```

```
git rev-list --disk-usage --objects --all --reflog
# total disk size used
du -c .git/objects/pack/*.pack .git/objects/??/*
# alternative to du: add up "size" and "size-pack" fields
git count-objects -v
```

- Report the disk size of each branch, not including objects used by the current branch. This can find outliers that are contributing to a bloated repository size (e.g., because somebody accidentally committed large build artifacts).

```
git for-each-ref --format='%(refname)' |
while read branch
do
        size=$(git rev-list --disk-usage --objects HEAD..$branch)
        echo "$size $branch"
done |
sort -n
```

- Compare the on-disk size of branches in one group of refs, excluding another. If you co-mingle objects from multiple remotes in a single repository, this can show which remotes are contributing to the repository size (taking the size of *origin* as a baseline).

```
git rev-list --disk-usage --objects --remotes=$suspect --not --
remotes=origin
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.121. git-rev-parse(1)

2024-07-02
Revision History
2024-07-02

NAME

git-rev-parse - Pick out and massage parameters

SYNOPSIS

git rev-parse [<options>] <arg>…

DESCRIPTION

Many Git porcelainish commands take a mixture of flags (i.e. parameters that begin with a dash -) and parameters meant for the underlying *git rev-list* command they use internally and flags and parameters for the other commands they use downstream of *git rev-list*. The primary purpose of this command is to allow calling programs to distinguish between them. There are a few other operation modes that have nothing to do with the above "help parse command line options".

Unless otherwise specified, most of the options and operation modes require you to run this command inside a git repository or a working tree that is under the control of a git repository, and will give you a fatal error otherwise.

OPTIONS

1. Operation Modes

Each of these options must appear first on the command line.

--parseopt

Use *git rev-parse* in option parsing mode (see PARSEOPT section below). The command in this mode can be used outside a repository or a working tree controlled by a repository.

--sq-quote

Use *git rev-parse* in shell quoting mode (see SQ-QUOTE section below). In contrast to the *--sq* option below, this mode only does quoting. Nothing else is done to command input. The command in this mode can be used outside a repository or a working tree controlled by a repository.

2. Options for --parseopt

--keep-dashdash

Only meaningful in *--parseopt* mode. Tells the option parser to echo out the first -- met instead of skipping it.

--stop-at-non-option

Only meaningful in *--parseopt* mode. Lets the option parser stop at the first non-option argument. This can be used to parse sub-commands that take options themselves.

--stuck-long

Only meaningful in *--parseopt* mode. Output the options in their long form if available, and with their arguments stuck.

3. Options for Filtering

--revs-only

Do not output flags and parameters not meant for *git rev-list* command.

--no-revs

Do not output flags and parameters meant for *git rev-list* command.

--flags

Do not output non-flag parameters.

--no-flags

Do not output flag parameters.

4. Options for Output

--default <arg>

If there is no parameter given by the user, use *<arg>* instead.

--prefix <arg>

Behave as if *git rev-parse* was invoked from the *<arg>* subdirectory of the working tree. Any relative filenames are resolved as if they are prefixed by *<arg>* and will be printed in that form.

This can be used to convert arguments to a command run in a subdirectory so that they can still be used after moving to the top-level of the repository. For example:

```
prefix=$(git rev-parse --show-prefix)
cd "$(git rev-parse --show-toplevel)"
# rev-parse provides the -- needed for 'set'
```

```
eval "set $(git rev-parse --sq --prefix "$prefix" -- "$@")"
```

--verify

Verify that exactly one parameter is provided, and that it can be turned into a raw 20-byte SHA-1 that can be used to access the object database. If so, emit it to the standard output; otherwise, error out.

If you want to make sure that the output actually names an object in your object database and/or can be used as a specific type of object you require, you can add the *^{type}* peeling operator to the parameter. For example, *git rev-parse "$VAR^{commit}"* will make sure *$VAR* names an existing object that is a commit-ish (i.e. a commit, or an annotated tag that points at a commit). To make sure that *$VAR* names an existing object of any type, *git rev-parse "$VAR^{object}"* can be used.

Note that if you are verifying a name from an untrusted source, it is wise to use *--end-of-options* so that the name argument is not mistaken for another option.

-q , --quiet

Only meaningful in *--verify* mode. Do not output an error message if the first argument is not a valid object name; instead exit with non-zero status silently. SHA-1s for valid object names are printed to stdout on success.

--sq

Usually the output is made one line per flag and parameter. This option makes output a single line, properly quoted for consumption by shell. Useful when you expect your parameter to contain whitespaces and newlines (e.g. when using pickaxe *-S* with *git diff-**). In contrast to the *--sq-quote* option, the command input is still interpreted as usual.

--short[=<length>]

Same as *--verify* but shortens the object name to a unique prefix with at least *length* characters. The minimum length is 4, the default is the effective value of the *core.abbrev* configuration variable (see Section G.3.29, “git-config(1)”).

--not

When showing object names, prefix them with ^ and strip ^ prefix from the object names that already have one.

--abbrev-ref[=(strict|loose)]

A non-ambiguous short name of the objects name. The option core.warnAmbiguousRefs is used to select the strict abbreviation mode.

--symbolic

Usually the object names are output in SHA-1 form (with possible ^ prefix); this option makes them output in a form as close to the original input as possible.

--symbolic-full-name

This is similar to --symbolic, but it omits input that are not refs (i.e. branch or tag names; or more explicitly disambiguating "heads/master" form, when you want to name the "master" branch when there is an unfortunately named tag "master"), and shows them as full refnames (e.g. "refs/heads/master").

--output-object-format=(sha1|sha256|storage)

Allow oids to be input from any object format that the current repository supports.

```
Specifying "sha1" translates if necessary and returns a sha1 oid.

Specifying "sha256" translates if necessary and returns a sha256 oid.
```

```
Specifying "storage" translates if necessary and returns an oid in
encoded in the storage hash algorithm.
```

5. Options for Objects

--all

Show all refs found in *refs/*.

--branches[=<pattern>] , --tags[=<pattern>] , --remotes[=<pattern>]

Show all branches, tags, or remote-tracking branches, respectively (i.e., refs found in *refs/heads*, *refs/tags*, or *refs/remotes*, respectively).

If a *pattern* is given, only refs matching the given shell glob are shown. If the pattern does not contain a globbing character (*?*, ***, or *[*), it is turned into a prefix match by appending */**.

--glob=<pattern>

Show all refs matching the shell glob pattern *pattern*. If the pattern does not start with *refs/*, this is automatically prepended. If the pattern does not contain a globbing character (*?*, ***, or *[*), it is turned into a prefix match by appending */**.

--exclude=<glob-pattern>

Do not include refs matching *<glob-pattern>* that the next *--all*, *--branches*, *--tags*, *--remotes*, or *--glob* would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next *--all*, *--branches*, *--tags*, *--remotes*, or *--glob* option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with *refs/heads*, *refs/tags*, or *refs/remotes* when applied to *--branches*, *--tags*, or *--remotes*, respectively, and they must begin with *refs/* when applied to *--glob* or *--all*. If a trailing */** is intended, it must be given explicitly.

--exclude-hidden=(fetch|receive|uploadpack)

Do not include refs that would be hidden by *git-fetch*, *git-receive-pack* or *git-upload-pack* by consulting the appropriate *fetch.hideRefs*, *receive.hideRefs* or *uploadpack.hideRefs* configuration along with *transfer.hideRefs* (see Section G.3.29, “git-config(1)”). This option affects the next pseudo-ref option *--all* or *--glob* and is cleared after processing them.

--disambiguate=<prefix>

Show every object whose name begins with the given prefix. The <prefix> must be at least 4 hexadecimal digits long to avoid listing each and every object in the repository by mistake.

6. Options for Files

--local-env-vars

List the GIT_* environment variables that are local to the repository (e.g. GIT_DIR or GIT_WORK_TREE, but not GIT_EDITOR). Only the names of the variables are listed, not their value, even if they are set.

--path-format=(absolute|relative)

Controls the behavior of certain other options. If specified as absolute, the paths printed by those options will be absolute and canonical. If specified as relative, the paths will be relative to the current working directory if that is possible. The default is option specific.

This option may be specified multiple times and affects only the arguments that follow it on the command line, either to the end of the command line or the next instance of this option.

The following options are modified by *--path-format*:

--git-dir

Show *$GIT_DIR* if defined. Otherwise show the path to the .git directory. The path shown, when relative, is relative to the current working directory.

If *$GIT_DIR* is not defined and the current directory is not detected to lie in a Git repository or work tree print a message to stderr and exit with nonzero status.

--git-common-dir

Show *$GIT_COMMON_DIR* if defined, else *$GIT_DIR*.

--resolve-git-dir <path>

Check if <path> is a valid repository or a gitfile that points at a valid repository, and print the location of the repository. If <path> is a gitfile then the resolved path to the real repository is printed.

--git-path <path>

Resolve "$GIT_DIR/<path>" and takes other path relocation variables such as $GIT_OBJECT_DIRECTORY, $GIT_INDEX_FILE… into account. For example, if $GIT_OBJECT_DIRECTORY is set to /foo/bar then "git rev-parse --git-path objects/abc" returns /foo/bar/abc.

--show-toplevel

Show the (by default, absolute) path of the top-level directory of the working tree. If there is no working tree, report an error.

--show-superproject-working-tree

Show the absolute path of the root of the superproject's working tree (if exists) that uses the current repository as its submodule. Outputs nothing if the current repository is not used as a submodule by any project.

--shared-index-path

Show the path to the shared index file in split index mode, or empty if not in split-index mode.

The following options are unaffected by *--path-format*:

--absolute-git-dir

Like *--git-dir*, but its output is always the canonicalized absolute path.

--is-inside-git-dir

When the current working directory is below the repository directory print "true", otherwise "false".

--is-inside-work-tree

When the current working directory is inside the work tree of the repository print "true", otherwise "false".

--is-bare-repository

When the repository is bare print "true", otherwise "false".

--is-shallow-repository

When the repository is shallow print "true", otherwise "false".

--show-cdup

When the command is invoked from a subdirectory, show the path of the top-level directory relative to the current directory (typically a sequence of "../", or an empty string).

--show-prefix

When the command is invoked from a subdirectory, show the path of the current directory relative to the top-level directory.

--show-object-format[=(storage|input|output)]

Show the object format (hash algorithm) used for the repository for storage inside the *.git* directory, input, or output. For input, multiple algorithms may be printed, space-separated. If not specified, the default is "storage".

--show-ref-format

Show the reference storage format used for the repository.

7. Other Options

--since=<datestring> , --after=<datestring>

Parse the date string, and output the corresponding --max-age= parameter for *git rev-list*.

--until=<datestring> , --before=<datestring>

Parse the date string, and output the corresponding --min-age= parameter for *git rev-list*.

<arg>…

Flags and parameters to be parsed.

SPECIFYING REVISIONS

A revision parameter *<rev>* typically, but not necessarily, names a commit object. It uses what is called an *extended SHA-1* syntax. Here are various ways to spell object names. The ones listed near the end of this list name trees and blobs contained in a commit.

> **Note**
>
> This document shows the "raw" syntax as seen by git. The shell and other UIs might require additional quoting to protect special characters and to avoid word splitting.

<sha1>, e.g. *dae86e1950b1277e545cee180551750029cfe735*, *dae86e*

The full SHA-1 object name (40-byte hexadecimal string), or a leading substring that is unique within the repository. E.g. dae86e1950b1277e545cee180551750029cfe735 and dae86e both name the same commit object if there is no other object in your repository whose object name starts with dae86e.

<describeOutput>, e.g. *v1.7.4.2-679-g3bee7fb*

Output from *git describe*; i.e. a closest tag, optionally followed by a dash and a number of commits, followed by a dash, a *g*, and an abbreviated object name.

<refname>, e.g. *master*, *heads/master*, *refs/heads/master*

A symbolic ref name. E.g. *master* typically means the commit object referenced by *refs/heads/master*. If you happen to have both *heads/master* and *tags/master*, you can explicitly say *heads/master* to tell Git which one you mean. When ambiguous, a *<refname>* is disambiguated by taking the first match in the following rules:

1. If *$GIT_DIR/<refname>* exists, that is what you mean (this is usually useful only for *HEAD*, *FETCH_HEAD*, *ORIG_HEAD*, *MERGE_HEAD*, *REBASE_HEAD*, *REVERT_HEAD*, *CHERRY_PICK_HEAD*, *BISECT_HEAD* and *AUTO_MERGE*);

2. otherwise, *refs/<refname>* if it exists;

3. otherwise, *refs/tags/<refname>* if it exists;
4. otherwise, *refs/heads/<refname>* if it exists;
5. otherwise, *refs/remotes/<refname>* if it exists;
6. otherwise, *refs/remotes/<refname>/HEAD* if it exists.

HEAD

names the commit on which you based the changes in the working tree.

FETCH_HEAD

records the branch which you fetched from a remote repository with your last *git fetch* invocation.

ORIG_HEAD

is created by commands that move your *HEAD* in a drastic way (*git am*, *git merge*, *git rebase*, *git reset*), to record the position of the *HEAD* before their operation, so that you can easily change the tip of the branch back to the state before you ran them.

MERGE_HEAD

records the commit(s) which you are merging into your branch when you run *git merge*.

REBASE_HEAD

during a rebase, records the commit at which the operation is currently stopped, either because of conflicts or an *edit* command in an interactive rebase.

REVERT_HEAD

records the commit which you are reverting when you run *git revert*.

CHERRY_PICK_HEAD

records the commit which you are cherry-picking when you run *git cherry-pick*.

BISECT_HEAD

records the current commit to be tested when you run *git bisect --no-checkout*.

AUTO_MERGE

records a tree object corresponding to the state the *ort* merge strategy wrote to the working tree when a merge operation resulted in conflicts.

Note that any of the *refs/** cases above may come either from the *$GIT_DIR/refs* directory or from the *$GIT_DIR/packed-refs* file. While the ref name encoding is unspecified, UTF-8 is preferred as some output processing may assume ref names in UTF-8.

@

@ alone is a shortcut for *HEAD*.

[<refname>]@{<date>}, e.g. *master@{yesterday}*, *HEAD@{5 minutes ago}*

A ref followed by the suffix @ with a date specification enclosed in a brace pair (e.g. *{yesterday}*, *{1 month 2 weeks 3 days 1 hour 1 second ago}* or *{1979-02-26 18:30:00}*) specifies the value of the ref at a prior point in time. This suffix may only be used immediately following a ref name and the ref must have an existing log (*$GIT_DIR/logs/<ref>*). Note that this looks up the state of your **local** ref at a given time; e.g., what was in your local *master* branch last week. If you want to look at commits made during certain times, see *--since* and *--until*.

<refname>@{<n>}, e.g. *master@{1}*

A ref followed by the suffix @ with an ordinal specification enclosed in a brace pair (e.g. *{1}*, *{15}*) specifies the n-th prior value of that ref. For example *master@{1}* is the immediate prior value of *master* while *master@{5}* is the 5th prior value of *master*. This suffix may only be used immediately following a ref name and the ref must have an existing log (*$GIT_DIR/logs/<refname>*).

@{<n>}, e.g. *@{1}*

You can use the @ construct with an empty ref part to get at a reflog entry of the current branch. For example, if you are on branch *blabla* then *@{1}* means the same as *blabla@{1}*.

@{-<n>}, e.g. *@{-1}*

The construct *@{-<n>}* means the <n>th branch/commit checked out before the current one.

[<branchname>]@{upstream}, e.g. *master@{upstream}*, *@{u}*

A branch B may be set up to build on top of a branch X (configured with *branch.<name>.merge*) at a remote R (configured with the branch X taken from remote R, typically found at *refs/remotes/R/X*.

[<branchname>]@{push}, e.g. *master@{push}*, *@{push}*

The suffix *@{push}* reports the branch "where we would push to" if *git push* were run while *branchname* was checked out (or the current *HEAD* if no branchname is specified). Like for *@{upstream}*, we report the remote-tracking branch that corresponds to that branch at the remote.

Here's an example to make it more clear:

```
$ git config push.default current
$ git config remote.pushdefault myfork
$ git switch -c mybranch origin/master

$ git rev-parse --symbolic-full-name @{upstream}
refs/remotes/origin/master

$ git rev-parse --symbolic-full-name @{push}
refs/remotes/myfork/mybranch
```

Note in the example that we set up a triangular workflow, where we pull from one location and push to another. In a non-triangular workflow, *@{push}* is the same as *@{upstream}*, and there is no need for it.

This suffix is also accepted when spelled in uppercase, and means the same thing no matter the case.

<rev>^[<n>], e.g. *HEAD^*, *v1.5.1^0*

A suffix ^ to a revision parameter means the first parent of that commit object. ^*<n>* means the <n>th parent (i.e. *<rev>^* is equivalent to *<rev>^1*). As a special rule, *<rev>^0* means the commit itself and is used when *<rev>* is the object name of a tag object that refers to a commit object.

<rev>~[<n>], e.g. *HEAD~*, *master~3*

A suffix ~ to a revision parameter means the first parent of that commit object. A suffix ~*<n>* to a revision parameter means the commit object that is the <n>th generation ancestor of the named commit object, following only the first parents. I.e. *<rev>~3* is equivalent to *<rev>^^^* which is equivalent to *<rev>^1^1^1*. See below for an illustration of the usage of this form.

<rev>^{<type>}, e.g. *v0.99.8^{commit}*

A suffix ^ followed by an object type name enclosed in brace pair means dereference the object at *<rev>* recursively until an object of type *<type>* is found or the object cannot be dereferenced anymore (in which

case, barf). For example, if *<rev>* is a commit-ish, *<rev>^{commit}* describes the corresponding commit object. Similarly, if *<rev>* is a tree-ish, *<rev>^{tree}* describes the corresponding tree object. *<rev>^0* is a short-hand for *<rev>^{commit}*.

<rev>^{object} can be used to make sure *<rev>* names an object that exists, without requiring *<rev>* to be a tag, and without dereferencing *<rev>*; because a tag is already an object, it does not have to be dereferenced even once to get to an object.

<rev>^{tag} can be used to ensure that *<rev>* identifies an existing tag object.

<rev>^{}, e.g. *v0.99.8^{}*

A suffix ^ followed by an empty brace pair means the object could be a tag, and dereference the tag recursively until a non-tag object is found.

<rev>^{/<text>}, e.g. *HEAD^{/fix nasty bug}*

A suffix ^ to a revision parameter, followed by a brace pair that contains a text led by a slash, is the same as the *:/fix nasty bug* syntax below except that it returns the youngest matching commit which is reachable from the *<rev>* before ^.

:/<text>, e.g. *:/fix nasty bug*

A colon, followed by a slash, followed by a text, names a commit whose commit message matches the specified regular expression. This name returns the youngest matching commit which is reachable from any ref, including HEAD. The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. *:/^foo*. The special sequence *:/!* is reserved for modifiers to what is matched. *:/!-foo* performs a negative match, while *:/!!foo* matches a literal *!* character, followed by *foo*. Any other sequence beginning with *:/!* is reserved for now. Depending on the given text, the shell's word splitting rules might require additional quoting.

<rev>:<path>, e.g. *HEAD:README*, *master:./README*

A suffix *:* followed by a path names the blob or tree at the given path in the tree-ish object named by the part before the colon. A path starting with *./* or *../* is relative to the current working directory. The given path will be converted to be relative to the working tree's root directory. This is most useful to address a blob or tree from a commit or tree that has the same tree structure as the working tree.

:[<n>:]<path>, e.g. *:0:README*, *:README*

A colon, optionally followed by a stage number (0 to 3) and a colon, followed by a path, names a blob object in the index at the given path. A missing stage number (and the colon that follows it) names a stage 0 entry. During a merge, stage 1 is the common ancestor, stage 2 is the target branch's version (typically the current branch), and stage 3 is the version from the branch which is being merged.

Here is an illustration, by Jon Loeliger. Both commit nodes B and C are parents of commit node A. Parent commits are ordered left-to-right.

```
G   H   I   J
 \ /     \ /
  D   E   F
   \  |  / \
    \ | /   |
     \|/    |
      B     C
       \   /
        \ /
         A

A =      = A^0
B = A^   = A^1     = A~1
```

```
C =      = A^2
D = A^^  = A^1^1   = A~2
E = B^2  = A^^2
F = B^3  = A^^3
G = A^^^ = A^1^1^1 = A~3
H = D^2  = B^^2    = A^^^2  = A~2^2
I = F^   = B^3^    = A^^3^
J = F^2  = B^3^2   = A^^3^2
```

SPECIFYING RANGES

History traversing commands such as *git log* operate on a set of commits, not just a single commit.

For these commands, specifying a single revision, using the notation described in the previous section, means the set of commits *reachable* from the given commit.

Specifying several revisions means the set of commits reachable from any of the given commits.

A commit's reachable set is the commit itself and the commits in its ancestry chain.

There are several notations to specify a set of connected commits (called a "revision range"), illustrated below.

1. Commit Exclusions

^<*rev*> (caret) Notation

To exclude commits reachable from a commit, a prefix ^ notation is used. E.g. ^*r1 r2* means commits reachable from *r2* but exclude the ones reachable from *r1* (i.e. *r1* and its ancestors).

2. Dotted Range Notations

The .. (two-dot) Range Notation

The ^*r1 r2* set operation appears so often that there is a shorthand for it. When you have two commits *r1* and *r2* (named according to the syntax explained in SPECIFYING REVISIONS above), you can ask for commits that are reachable from r2 excluding those that are reachable from r1 by ^*r1 r2* and it can be written as *r1..r2*.

The ... (three-dot) Symmetric Difference Notation

A similar notation *r1...r2* is called symmetric difference of *r1* and *r2* and is defined as *r1 r2 --not $(git merge-base --all r1 r2)*. It is the set of commits that are reachable from either one of *r1* (left side) or *r2* (right side) but not from both.

In these two shorthand notations, you can omit one end and let it default to HEAD. For example, *origin..* is a shorthand for *origin..HEAD* and asks "What did I do since I forked from the origin branch?" Similarly, *..origin* is a shorthand for *HEAD..origin* and asks "What did the origin do since I forked from them?" Note that .. would mean *HEAD..HEAD* which is an empty range that is both reachable and unreachable from HEAD.

Commands that are specifically designed to take two distinct ranges (e.g. "git range-diff R1 R2" to compare two ranges) do exist, but they are exceptions. Unless otherwise noted, all "git" commands that operate on a set of commits work on a single revision range. In other words, writing two "two-dot range notation" next to each other, e.g.

```
$ git log A..B C..D
```

does **not** specify two revision ranges for most commands. Instead it will name a single connected set of commits, i.e. those that are reachable from either B or D but are reachable from neither A or C. In a linear history like this:

```
---A---B---o---o---C---D
```

because A and B are reachable from C, the revision range specified by these two dotted ranges is a single commit D.

3. Other <rev>^ Parent Shorthand Notations

Three other shorthands exist, particularly useful for merge commits, for naming a set that is formed by a commit and its parent commits.

The *r1^@* notation means all parents of *r1*.

The *r1^!* notation includes commit *r1* but excludes all of its parents. By itself, this notation denotes the single commit *r1*.

The *<rev>^-[<n>]* notation includes *<rev>* but excludes the <n>th parent (i.e. a shorthand for *<rev>^<n>..<rev>*), with *<n>* = 1 if not given. This is typically useful for merge commits where you can just pass *<commit>^-* to get all the commits in the branch that was merged in merge commit *<commit>* (including *<commit>* itself).

While *<rev>^<n>* was about specifying a single commit parent, these three notations also consider its parents. For example you can say *HEAD^2^@*, however you cannot say *HEAD^@^2*.

Revision Range Summary

<rev>

Include commits that are reachable from <rev> (i.e. <rev> and its ancestors).

^*<rev>*

Exclude commits that are reachable from <rev> (i.e. <rev> and its ancestors).

<rev1>..<rev2>

Include commits that are reachable from <rev2> but exclude those that are reachable from <rev1>. When either <rev1> or <rev2> is omitted, it defaults to *HEAD*.

<rev1>...<rev2>

Include commits that are reachable from either <rev1> or <rev2> but exclude those that are reachable from both. When either <rev1> or <rev2> is omitted, it defaults to *HEAD*.

<rev>^@, e.g. *HEAD^@*

A suffix ^ followed by an at sign is the same as listing all parents of *<rev>* (meaning, include anything reachable from its parents, but not the commit itself).

<rev>^!, e.g. *HEAD^!*

A suffix ^ followed by an exclamation mark is the same as giving commit *<rev>* and all its parents prefixed with ^ to exclude them (and their ancestors).

<rev>^-*<n>*, e.g. *HEAD^-*, *HEAD^-2*

Equivalent to *<rev>^<n>..<rev>*, with *<n>* = 1 if not given.

Here are a handful of examples using the Loeliger illustration above, with each step in the notation's expansion and selection carefully spelt out:

```
Args   Expanded arguments    Selected commits
D                            G H D
D F                          G H I J D F
^G D                         H D
^D B                         E I J F B
^D B C                       E I J F B C
C                            I J F C
```

```
B..C    = ^B C                  C
B...C   = B ^F C                G H D E B C
B^-     = B^..B
        = ^B^1 B                E I J F B
C^@     = C^1
        = F                     I J F
B^@     = B^1 B^2 B^3
        = D E F                 D G H E F I J
C^!     = C ^C^@
        = C ^C^1
        = C ^F                  C
B^!     = B ^B^@
        = B ^B^1 ^B^2 ^B^3
        = B ^D ^E ^F            B
F^! D   = F ^I ^J D             G H D F
```

PARSEOPT

In *--parseopt* mode, *git rev-parse* helps massaging options to bring to shell scripts the same facilities C builtins have. It works as an option normalizer (e.g. splits single switches aggregate values), a bit like *getopt(1)* does.

It takes on the standard input the specification of the options to parse and understand, and echoes on the standard output a string suitable for *sh(1) eval* to replace the arguments with normalized ones. In case of error, it outputs usage on the standard error stream, and exits with code 129.

Note: Make sure you quote the result when passing it to *eval*. See below for an example.

1. Input Format

git rev-parse --parseopt input format is fully text based. It has two parts, separated by a line that contains only --. The lines before the separator (should be one or more) are used for the usage. The lines after the separator describe the options.

Each line of options has this format:

`<opt-spec><flags>*<arg-hint>? SP+ help LF`

<opt-spec>

its format is the short option character, then the long option name separated by a comma. Both parts are not required, though at least one is necessary. May not contain any of the *<flags>* characters. *h,help*, *dry-run* and *f* are examples of correct *<opt-spec>*.

<flags>

<flags> are of *, =, ? or !.

- Use = if the option takes an argument.
- Use ? to mean that the option takes an optional argument. You probably want to use the *--stuck-long* mode to be able to unambiguously parse the optional argument.
- Use * to mean that this option should not be listed in the usage generated for the *-h* argument. It's shown for *--help-all* as documented in Section G.4.1, “gitcli(7)”.
- Use ! to not make the corresponding negated long option available.

<arg-hint>

<arg-hint>, if specified, is used as a name of the argument in the help output, for options that take arguments. *<arg-hint>* is terminated by the first whitespace. It is customary to use a dash to separate words in a multi-word argument hint.

The remainder of the line, after stripping the spaces, is used as the help associated with the option.

Blank lines are ignored, and lines that don't match this specification are used as option group headers (start the line with a space to create such lines on purpose).

2. Example

```
OPTS_SPEC="\
some-command [<options>] <args>...

some-command does foo and bar!
--
h,help!   show the help

foo       some nifty option --foo
bar=      some cool option --bar with an argument
baz=arg   another cool option --baz with a named argument
qux?path  qux may take a path argument but has meaning by itself

  An option group Header
C?        option C with an optional argument"

eval "$(echo "$OPTS_SPEC" | git rev-parse --parseopt -- "$@" || echo exit
 $?)"
```

3. Usage text

When *"$@"* is *-h* or *--help* in the above example, the following usage text would be shown:

```
usage: some-command [<options>] <args>...

    some-command does foo and bar!

    -h, --help            show the help
    --[no-]foo            some nifty option --foo
    --[no-]bar ...        some cool option --bar with an argument
    --[no-]baz <arg>      another cool option --baz with a named argument
    --[no-]qux[=<path>]   qux may take a path argument but has meaning by
 itself

An option group Header
    -C[...]               option C with an optional argument
```

SQ-QUOTE

In *--sq-quote* mode, *git rev-parse* echoes on the standard output a single line suitable for *sh(1) eval*. This line is made by normalizing the arguments following *--sq-quote*. Nothing other than quoting the arguments is done.

If you want command input to still be interpreted as usual by *git rev-parse* before the output is shell quoted, see the *--sq* option.

1. Example

```
$ cat >your-git-script.sh <<\EOF
#!/bin/sh
args=$(git rev-parse --sq-quote "$@")   # quote user-supplied arguments
command="git frotz -n24 $args"          # and use it inside a handcrafted
                                        # command line
```

```
eval "$command"
EOF

$ sh your-git-script.sh "a b'c"
```

EXAMPLES

- Print the object name of the current commit:

  ```
  $ git rev-parse --verify HEAD
  ```

- Print the commit object name from the revision in the $REV shell variable:

  ```
  $ git rev-parse --verify --end-of-options $REV^{commit}
  ```

  This will error out if $REV is empty or not a valid revision.

- Similar to above:

  ```
  $ git rev-parse --default master --verify --end-of-options $REV
  ```

  but if $REV is empty, the commit object name from master will be printed.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.122. git-revert(1)

2024-07-02
Revision History
2024-07-02

NAME

git-revert - Revert some existing commits

SYNOPSIS

git revert [--[no-]edit] [-n] [-m <parent-number>] [-s] [-S[<keyid>]] <commit>…
git revert (--continue | --skip | --abort | --quit)

DESCRIPTION

Given one or more existing commits, revert the changes that the related patches introduce, and record some new commits that record them. This requires your working tree to be clean (no modifications from the HEAD commit).

Note: *git revert* is used to record some new commits to reverse the effect of some earlier commits (often only a faulty one). If you want to throw away all uncommitted changes in your working directory, you should see Section G.3.118, “git-reset(1)”, particularly the *--hard* option. If you want to extract specific files as they were in another commit, you should see Section G.3.119, “git-restore(1)”, specifically the *--source* option. Take care with these alternatives as both will discard uncommitted changes in your working directory.

See "Reset, restore and revert" in Section G.3.1, “git(1)” for the differences between the three commands.

OPTIONS

<commit>…

Commits to revert. For a more complete list of ways to spell commit names, see Section G.4.14, “gitrevisions(7)”. Sets of commits can also be given but no traversal is done by default, see Section G.3.120, “git-rev-list(1)” and its *--no-walk* option.

-e , --edit

With this option, *git revert* will let you edit the commit message prior to committing the revert. This is the default if you run the command from a terminal.

-m parent-number , --mainline parent-number

Usually you cannot revert a merge because you do not know which side of the merge should be considered the mainline. This option specifies the parent number (starting from 1) of the mainline and allows revert to reverse the change relative to the specified parent.

Reverting a merge commit declares that you will never want the tree changes brought in by the merge. As a result, later merges will only bring in tree changes introduced by commits that are not ancestors of the previously reverted merge. This may or may not be what you want.

See the *revert-a-faulty-merge How-To* [https://www.kernel.org/pub/software/scm/git/docs/howto/revert-a-faulty-merge.html] for more details.

--no-edit

With this option, *git revert* will not start the commit message editor.

--cleanup=<mode>

This option determines how the commit message will be cleaned up before being passed on to the commit machinery. See Section G.3.28, “git-commit(1)” for more details. In particular, if the *<mode>* is given a value of *scissors*, scissors will be appended to *MERGE_MSG* before being passed on in the case of a conflict.

-n , --no-commit

Usually the command automatically creates some commits with commit log messages stating which commits were reverted. This flag applies the changes necessary to revert the named commits to your working tree and the index, but does not make the commits. In addition, when this option is used, your index does not have to match the HEAD commit. The revert is done against the beginning state of your index.

This is useful when reverting more than one commits' effect to your index in a row.

-S[<keyid>] , --gpg-sign[=<keyid>] , --no-gpg-sign

GPG-sign commits. The *keyid* argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space. *--no-gpg-sign* is useful to countermand both *commit.gpgSign* configuration variable, and earlier *--gpg-sign*.

-s , --signoff

Add a *Signed-off-by* trailer at the end of the commit message. See the signoff option in Section G.3.28, “git-commit(1)” for more information.

--strategy=<strategy>

Use the given merge strategy. Should only be used once. See the MERGE STRATEGIES section in Section G.3.86, “git-merge(1)” for details.

-X<option> , --strategy-option=<option>

Pass the merge strategy-specific option through to the merge strategy. See Section G.3.86, “git-merge(1)” for details.

--rerere-autoupdate , --no-rerere-autoupdate

After the rerere mechanism reuses a recorded resolution on the current conflict to update the files in the working tree, allow it to also update the index with the result of resolution. *--no-rerere-autoupdate* is a good

way to double-check what *rerere* did and catch potential mismerges, before committing the result to the index with a separate *git add*.

--reference

Instead of starting the body of the log message with "This reverts <full-object-name-of-the-commit-being-reverted>.", refer to the commit using "--pretty=reference" format (cf. Section G.3.74, “git-log(1)”). The *revert.reference* configuration variable can be used to enable this option by default.

SEQUENCER SUBCOMMANDS

--continue

Continue the operation in progress using the information in *.git/sequencer*. Can be used to continue after resolving conflicts in a failed cherry-pick or revert.

--skip

Skip the current commit and continue with the rest of the sequence.

--quit

Forget about the current operation in progress. Can be used to clear the sequencer state after a failed cherry-pick or revert.

--abort

Cancel the operation and return to the pre-sequence state.

EXAMPLES

git revert HEAD~3

Revert the changes specified by the fourth last commit in HEAD and create a new commit with the reverted changes.

git revert -n master~5..master~2

Revert the changes done by commits from the fifth last commit in master (included) to the third last commit in master (included), but do not create any commit with the reverted changes. The revert only modifies the working tree and the index.

DISCUSSION

While git creates a basic commit message automatically, it is *strongly* recommended to explain why the original commit is being reverted. In addition, repeatedly reverting reverts will result in increasingly unwieldy subject lines, for example *Reapply "Reapply "<original-subject>""*. Please consider rewording these to be shorter and more unique.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

revert.reference

Setting this variable to true makes *git revert* behave as if the *--reference* option is given.

SEE ALSO

Section G.3.20, “git-cherry-pick(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.123. git-rm(1)

2024-07-02
Revision History
2024-07-02

NAME

git-rm - Remove files from the working tree and from the index

SYNOPSIS

git rm [-f | --force] [-n] [-r] [--cached] [--ignore-unmatch]
[--quiet] [--pathspec-from-file=<file> [--pathspec-file-nul]]
[--] [<pathspec>…]

DESCRIPTION

Remove files matching pathspec from the index, or from the working tree and the index. *git rm* will not remove a file from just your working directory. (There is no option to remove a file only from the working tree and yet keep it in the index; use */bin/rm* if you want to do that.) The files being removed have to be identical to the tip of the branch, and no updates to their contents can be staged in the index, though that default behavior can be overridden with the *-f* option. When *--cached* is given, the staged content has to match either the tip of the branch or the file on disk, allowing the file to be removed from just the index. When sparse-checkouts are in use (see Section G.3.135, “git-sparse-checkout(1)”), *git rm* will only remove paths within the sparse-checkout patterns.

OPTIONS

<pathspec>…

Files to remove. A leading directory name (e.g. *dir* to remove *dir/file1* and *dir/file2*) can be given to remove all files in the directory, and recursively all sub-directories, but this requires the *-r* option to be explicitly given.

The command removes only the paths that are known to Git.

File globbing matches across directory boundaries. Thus, given two directories *d* and *d2*, there is a difference between using *git rm 'd*'* and *git rm 'd/*'*, as the former will also remove all of directory *d2*.

For more details, see the *pathspec* entry in Section G.4.19, “gitglossary(7)”.

-f , --force

Override the up-to-date check.

-n , --dry-run

Don't actually remove any file(s). Instead, just show if they exist in the index and would otherwise be removed by the command.

-r

Allow recursive removal when a leading directory name is given.

--

This option can be used to separate command-line options from the list of files, (useful when filenames might be mistaken for command-line options).

--cached

Use this option to unstage and remove paths only from the index. Working tree files, whether modified or not, will be left alone.

--ignore-unmatch

Exit with a zero status even if no files matched.

--sparse

Allow updating index entries outside of the sparse-checkout cone. Normally, *git rm* refuses to update index entries whose paths do not fit within the sparse-checkout cone. See Section G.3.135, "git-sparse-checkout(1)" for more.

-q , --quiet

git rm normally outputs one line (in the form of an *rm* command) for each file removed. This option suppresses that output.

--pathspec-from-file=<file>

Pathspec is passed in <*file*> instead of commandline args. If <*file*> is exactly - then standard input is used. Pathspec elements are separated by LF or CR/LF. Pathspec elements can be quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, "git-config(1)"). See also *--pathspec-file-nul* and global *--literal-pathspecs*.

--pathspec-file-nul

Only meaningful with *--pathspec-from-file*. Pathspec elements are separated with NUL character and all other characters are taken literally (including newlines and quotes).

REMOVING FILES THAT HAVE DISAPPEARED FROM THE FILESYSTEM

There is no option for *git rm* to remove from the index only the paths that have disappeared from the filesystem. However, depending on the use case, there are several ways that can be done.

1. Using git commit -a

If you intend that your next commit should record all modifications of tracked files in the working tree and record all removals of files that have been removed from the working tree with *rm* (as opposed to *git rm*), use *git commit -a*, as it will automatically notice and record all removals. You can also have a similar effect without committing by using *git add -u*.

2. Using git add -A

When accepting a new code drop for a vendor branch, you probably want to record both the removal of paths and additions of new paths as well as modifications of existing paths.

Typically you would first remove all tracked files from the working tree using this command:

```
git ls-files -z | xargs -0 rm -f
```

and then untar the new code in the working tree. Alternately you could *rsync* the changes into the working tree.

After that, the easiest way to record all removals, additions, and modifications in the working tree is:

```
git add -A
```

See Section G.3.2, "git-add(1)".

3. Other ways

If all you really want to do is to remove from the index the files that are no longer present in the working tree (perhaps because your working tree is dirty so that you cannot use *git commit -a*), use the following command:

```
git diff --name-only --diff-filter=D -z | xargs -0 git rm --cached
```

SUBMODULES

Only submodules using a gitfile (which means they were cloned with a Git version 1.7.8 or newer) will be removed from the work tree, as their repository lives inside the .git directory of the superproject. If a submodule (or one of those nested inside it) still uses a .git directory, *git rm* will move the submodules git directory into the superprojects git directory to protect the submodule's history. If it exists the submodule.<name> section in the Section G.4.10, “gitmodules(5)” file will also be removed and that file will be staged (unless --cached or -n are used).

A submodule is considered up to date when the HEAD is the same as recorded in the index, no tracked files are modified and no untracked files that aren't ignored are present in the submodule's work tree. Ignored files are deemed expendable and won't stop a submodule's work tree from being removed.

If you only want to remove the local checkout of a submodule from your work tree without committing the removal, use Section G.3.141, “git-submodule(1)” *deinit* instead. Also see Section G.4.15, “gitsubmodules(7)” for details on submodule removal.

EXAMPLES

git rm Documentation/.txt*

Removes all **.txt* files from the index that are under the *Documentation* directory and any of its subdirectories.

Note that the asterisk * is quoted from the shell in this example; this lets Git, and not the shell, expand the pathnames of files and subdirectories under the *Documentation/* directory.

git rm -f git-.sh*

Because this example lets the shell expand the asterisk (i.e. you are listing the files explicitly), it does not remove *subdir/git-foo.sh*.

BUGS

Each time a superproject update removes a populated submodule (e.g. when switching between commits before and after the removal) a stale submodule checkout will remain in the old location. Removing the old directory is only safe when it uses a gitfile, as otherwise the history of the submodule will be deleted too. This step will be obsolete when recursive submodule update has been implemented.

SEE ALSO

Section G.3.2, “git-add(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.124. git-send-email(1)

2024-07-02

Revision History

2024-07-02

NAME

git-send-email - Send a collection of patches as emails

SYNOPSIS

git send-email [<options>] (<file>|<directory>)…
git send-email [<options>] <format-patch-options>
git send-email --dump-aliases

DESCRIPTION

Takes the patches given on the command line and emails them out. Patches can be specified as files, directories (which will send all files in the directory), or directly as a revision list. In the last case, any format accepted by Section G.3.54, “git-format-patch(1)” can be passed to git send-email, as well as options understood by Section G.3.54, “git-format-patch(1)”.

The header of the email is configurable via command-line options. If not specified on the command line, the user will be prompted with a ReadLine enabled interface to provide the necessary information.

There are two formats accepted for patch files:

1. mbox format files

   This is what Section G.3.54, “git-format-patch(1)” generates. Most headers and MIME formatting are ignored.

2. The original format used by Greg Kroah-Hartman's *send_lots_of_email.pl* script

   This format expects the first line of the file to contain the "Cc:" value and the "Subject:" of the message as the second line.

OPTIONS

1. Composing

--annotate

Review and edit each patch you're about to send. Default is the value of *sendemail.annotate*. See the CONFIGURATION section for *sendemail.multiEdit*.

--bcc=<address>,…

Specify a "Bcc:" value for each email. Default is the value of *sendemail.bcc*.

This option may be specified multiple times.

--cc=<address>,…

Specify a starting "Cc:" value for each email. Default is the value of *sendemail.cc*.

This option may be specified multiple times.

--compose

Invoke a text editor (see GIT_EDITOR in Section G.3.152, “git-var(1)”) to edit an introductory message for the patch series.

When *--compose* is used, git send-email will use the From, To, Cc, Bcc, Subject, Reply-To, and In-Reply-To headers specified in the message. If the body of the message (what you type after the headers and a blank line) only contains blank (or Git: prefixed) lines, the summary won't be sent, but the headers mentioned above will be used unless they are removed.

Missing From or In-Reply-To headers will be prompted for.

See the CONFIGURATION section for *sendemail.multiEdit*.

--from=<address>

Specify the sender of the emails. If not specified on the command line, the value of the *sendemail.from* configuration option is used. If neither the command-line option nor *sendemail.from* are set, then the user will be prompted for the value. The default for the prompt will be the value of GIT_AUTHOR_IDENT, or GIT_COMMITTER_IDENT if that is not set, as returned by "git var -l".

--reply-to=<address>

Specify the address where replies from recipients should go to. Use this if replies to messages should go to another address than what is specified with the --from parameter.

--in-reply-to=<identifier>

Make the first mail (or all the mails with *--no-thread*) appear as a reply to the given Message-ID, which avoids breaking threads to provide a new patch series. The second and subsequent emails will be sent as replies according to the *--[no-]chain-reply-to* setting.

So for example when *--thread* and *--no-chain-reply-to* are specified, the second and subsequent patches will be replies to the first one like in the illustration below where *[PATCH v2 0/3]* is in reply to *[PATCH 0/2]*:

```
[PATCH 0/2] Here is what I did...
  [PATCH 1/2] Clean up and tests
  [PATCH 2/2] Implementation
  [PATCH v2 0/3] Here is a reroll
    [PATCH v2 1/3] Clean up
    [PATCH v2 2/3] New tests
    [PATCH v2 3/3] Implementation
```

Only necessary if --compose is also set. If --compose is not set, this will be prompted for.

--subject=<string>

Specify the initial subject of the email thread. Only necessary if --compose is also set. If --compose is not set, this will be prompted for.

--to=<address>,…

Specify the primary recipient of the emails generated. Generally, this will be the upstream maintainer of the project involved. Default is the value of the *sendemail.to* configuration value; if that is unspecified, and --to-cmd is not specified, this will be prompted for.

This option may be specified multiple times.

--8bit-encoding=<encoding>

When encountering a non-ASCII message or subject that does not declare its encoding, add headers/quoting to indicate it is encoded in <encoding>. Default is the value of the *sendemail.assume8bitEncoding*; if that is unspecified, this will be prompted for if any non-ASCII files are encountered.

Note that no attempts whatsoever are made to validate the encoding.

--compose-encoding=<encoding>

Specify encoding of compose message. Default is the value of the *sendemail.composeEncoding*; if that is unspecified, UTF-8 is assumed.

--transfer-encoding=(7bit|8bit|quoted-printable|base64|auto)

Specify the transfer encoding to be used to send the message over SMTP. 7bit will fail upon encountering a non-ASCII message. quoted-printable can be useful when the repository contains files that contain carriage returns, but makes the raw patch email file (as saved from a MUA) much harder to inspect manually. base64

is even more fool proof, but also even more opaque. auto will use 8bit when possible, and quoted-printable otherwise.

Default is the value of the *sendemail.transferEncoding* configuration value; if that is unspecified, default to *auto*.

--xmailer , --no-xmailer

Add (or prevent adding) the "X-Mailer:" header. By default, the header is added, but it can be turned off by setting the *sendemail.xmailer* configuration variable to *false*.

2. Sending

--envelope-sender=<address>

Specify the envelope sender used to send the emails. This is useful if your default address is not the address that is subscribed to a list. In order to use the *From* address, set the value to "auto". If you use the sendmail binary, you must have suitable privileges for the -f parameter. Default is the value of the *sendemail.envelopeSender* configuration variable; if that is unspecified, choosing the envelope sender is left to your MTA.

--sendmail-cmd=<command>

Specify a command to run to send the email. The command should be sendmail-like; specifically, it must support the *-i* option. The command will be executed in the shell if necessary. Default is the value of *sendemail.sendmailCmd*. If unspecified, and if --smtp-server is also unspecified, git-send-email will search for *sendmail* in */usr/sbin*, */usr/lib* and $PATH.

--smtp-encryption=<encryption>

Specify in what way encrypting begins for the SMTP connection. Valid values are *ssl* and *tls*. Any other value reverts to plain (unencrypted) SMTP, which defaults to port 25. Despite the names, both values will use the same newer version of TLS, but for historic reasons have these names. *ssl* refers to "implicit" encryption (sometimes called SMTPS), that uses port 465 by default. *tls* refers to "explicit" encryption (often known as STARTTLS), that uses port 25 by default. Other ports might be used by the SMTP server, which are not the default. Commonly found alternative port for *tls* and unencrypted is 587. You need to check your provider's documentation or your server configuration to make sure for your own case. Default is the value of *sendemail.smtpEncryption*.

--smtp-domain=<FQDN>

Specifies the Fully Qualified Domain Name (FQDN) used in the HELO/EHLO command to the SMTP server. Some servers require the FQDN to match your IP address. If not set, git send-email attempts to determine your FQDN automatically. Default is the value of *sendemail.smtpDomain*.

--smtp-auth=<mechanisms>

Whitespace-separated list of allowed SMTP-AUTH mechanisms. This setting forces using only the listed mechanisms. Example:

```
$ git send-email --smtp-auth="PLAIN LOGIN GSSAPI" ...
```

If at least one of the specified mechanisms matches the ones advertised by the SMTP server and if it is supported by the utilized SASL library, the mechanism is used for authentication. If neither *sendemail.smtpAuth* nor *--smtp-auth* is specified, all mechanisms supported by the SASL library can be used. The special value *none* maybe specified to completely disable authentication independently of *--smtp-user*

--smtp-pass[=<password>]

Password for SMTP-AUTH. The argument is optional: If no argument is specified, then the empty string is used as the password. Default is the value of *sendemail.smtpPass*, however *--smtp-pass* always overrides this value.

Furthermore, passwords need not be specified in configuration files or on the command line. If a username has been specified (with *--smtp-user* or a *sendemail.smtpUser*), but no password has been specified (with *--smtp-pass* or *sendemail.smtpPass*), then a password is obtained using *git-credential*.

--no-smtp-auth

Disable SMTP authentication. Short hand for *--smtp-auth=none*

--smtp-server=<host>

If set, specifies the outgoing SMTP server to use (e.g. *smtp.example.com* or a raw IP address). If unspecified, and if *--sendmail-cmd* is also unspecified, the default is to search for *sendmail* in */usr/sbin*, */usr/lib* and $PATH if such a program is available, falling back to *localhost* otherwise.

For backward compatibility, this option can also specify a full pathname of a sendmail-like program instead; the program must support the *-i* option. This method does not support passing arguments or using plain command names. For those use cases, consider using *--sendmail-cmd* instead.

--smtp-server-port=<port>

Specifies a port different from the default port (SMTP servers typically listen to smtp port 25, but may also listen to submission port 587, or the common SSL smtp port 465); symbolic port names (e.g. "submission" instead of 587) are also accepted. The port can also be set with the *sendemail.smtpServerPort* configuration variable.

--smtp-server-option=<option>

If set, specifies the outgoing SMTP server option to use. Default value can be specified by the *sendemail.smtpServerOption* configuration option.

The --smtp-server-option option must be repeated for each option you want to pass to the server. Likewise, different lines in the configuration files must be used for each option.

--smtp-ssl

Legacy alias for *--smtp-encryption ssl*.

--smtp-ssl-cert-path

Path to a store of trusted CA certificates for SMTP SSL/TLS certificate validation (either a directory that has been processed by *c_rehash*, or a single file containing one or more PEM format certificates concatenated together: see verify(1) -CAfile and -CApath for more information on these). Set it to an empty string to disable certificate verification. Defaults to the value of the *sendemail.smtpSSLCertPath* configuration variable, if set, or the backing SSL library's compiled-in default otherwise (which should be the best choice on most platforms).

--smtp-user=<user>

Username for SMTP-AUTH. Default is the value of *sendemail.smtpUser*; if a username is not specified (with *--smtp-user* or *sendemail.smtpUser*), then authentication is not attempted.

--smtp-debug=(0|1)

Enable (1) or disable (0) debug output. If enabled, SMTP commands and replies will be printed. Useful to debug TLS connection and authentication problems.

--batch-size=<num>

Some email servers (e.g. smtp.163.com) limit the number emails to be sent per session (connection) and this will lead to a failure when sending many messages. With this option, send-email will disconnect after sending $<num> messages and wait for a few seconds (see --relogin-delay) and reconnect, to work around such a

limit. You may want to use some form of credential helper to avoid having to retype your password every time this happens. Defaults to the *sendemail.smtpBatchSize* configuration variable.

--relogin-delay=<int>

Waiting $<int> seconds before reconnecting to SMTP server. Used together with --batch-size option. Defaults to the *sendemail.smtpReloginDelay* configuration variable.

3. Automating

--no-to , --no-cc , --no-bcc

Clears any list of "To:", "Cc:", "Bcc:" addresses previously set via config.

--no-identity

Clears the previously read value of *sendemail.identity* set via config, if any.

--to-cmd=<command>

Specify a command to execute once per patch file which should generate patch file specific "To:" entries. Output of this command must be single email address per line. Default is the value of *sendemail.toCmd* configuration value.

--cc-cmd=<command>

Specify a command to execute once per patch file which should generate patch file specific "Cc:" entries. Output of this command must be single email address per line. Default is the value of *sendemail.ccCmd* configuration value.

--header-cmd=<command>

Specify a command that is executed once per outgoing message and output RFC 2822 style header lines to be inserted into them. When the *sendemail.headerCmd* configuration variable is set, its value is always used. When --header-cmd is provided at the command line, its value takes precedence over the *sendemail.headerCmd* configuration variable.

--no-header-cmd

Disable any header command in use.

--[no-]chain-reply-to

If this is set, each email will be sent as a reply to the previous email sent. If disabled with "--no-chain-reply-to", all emails after the first will be sent as replies to the first email sent. When using this, it is recommended that the first file given be an overview of the entire patch series. Disabled by default, but the *sendemail.chainReplyTo* configuration variable can be used to enable it.

--identity=<identity>

A configuration identity. When given, causes values in the *sendemail.<identity>* subsection to take precedence over values in the *sendemail* section. The default identity is the value of *sendemail.identity*.

--[no-]signed-off-by-cc

If this is set, add emails found in the *Signed-off-by* trailer or Cc: lines to the cc list. Default is the value of *sendemail.signedOffByCc* configuration value; if that is unspecified, default to --signed-off-by-cc.

--[no-]cc-cover

If this is set, emails found in Cc: headers in the first patch of the series (typically the cover letter) are added to the cc list for each email set. Default is the value of *sendemail.ccCover* configuration value; if that is unspecified, default to --no-cc-cover.

--[no-]to-cover

If this is set, emails found in To: headers in the first patch of the series (typically the cover letter) are added to the to list for each email set. Default is the value of *sendemail.toCover* configuration value; if that is unspecified, default to --no-to-cover.

--suppress-cc=<category>

Specify an additional category of recipients to suppress the auto-cc of:

- *author* will avoid including the patch author.
- *self* will avoid including the sender.
- *cc* will avoid including anyone mentioned in Cc lines in the patch header except for self (use *self* for that).
- *bodycc* will avoid including anyone mentioned in Cc lines in the patch body (commit message) except for self (use *self* for that).
- *sob* will avoid including anyone mentioned in the Signed-off-by trailers except for self (use *self* for that).
- *misc-by* will avoid including anyone mentioned in Acked-by, Reviewed-by, Tested-by and other "-by" lines in the patch body, except Signed-off-by (use *sob* for that).
- *cccmd* will avoid running the --cc-cmd.
- *body* is equivalent to *sob* + *bodycc* + *misc-by*.
- *all* will suppress all auto cc values.

Default is the value of *sendemail.suppressCc* configuration value; if that is unspecified, default to *self* if --suppress-from is specified, as well as *body* if --no-signed-off-cc is specified.

--[no-]suppress-from

If this is set, do not add the From: address to the cc: list. Default is the value of *sendemail.suppressFrom* configuration value; if that is unspecified, default to --no-suppress-from.

--[no-]thread

If this is set, the In-Reply-To and References headers will be added to each email sent. Whether each mail refers to the previous email (*deep* threading per *git format-patch* wording) or to the first email (*shallow* threading) is governed by "--[no-]chain-reply-to".

If disabled with "--no-thread", those headers will not be added (unless specified with --in-reply-to). Default is the value of the *sendemail.thread* configuration value; if that is unspecified, default to --thread.

It is up to the user to ensure that no In-Reply-To header already exists when *git send-email* is asked to add it (especially note that *git format-patch* can be configured to do the threading itself). Failure to do so may not produce the expected result in the recipient's MUA.

4. Administering

--confirm=<mode>

Confirm just before sending:

- *always* will always confirm before sending
- *never* will never confirm before sending

- *cc* will confirm before sending when send-email has automatically added addresses from the patch to the Cc list
- *compose* will confirm before sending the first message when using --compose.
- *auto* is equivalent to *cc* + *compose*

Default is the value of *sendemail.confirm* configuration value; if that is unspecified, default to *auto* unless any of the suppress options have been specified, in which case default to *compose*.

--dry-run

Do everything except actually send the emails.

--[no-]format-patch

When an argument may be understood either as a reference or as a file name, choose to understand it as a format-patch argument (*--format-patch*) or as a file name (*--no-format-patch*). By default, when such a conflict occurs, git send-email will fail.

--quiet

Make git-send-email less verbose. One line per email should be all that is output.

--[no-]validate

Perform sanity checks on patches. Currently, validation means the following:

- Invoke the sendemail-validate hook if present (see Section G.4.7, “githooks(5)”).
- Warn of patches that contain lines longer than 998 characters unless a suitable transfer encoding (*auto*, *base64*, or *quoted-printable*) is used; this is due to SMTP limits as described by https://www.ietf.org/rfc/rfc5322.txt.

Default is the value of *sendemail.validate*; if this is not set, default to *--validate*.

--force

Send emails even if safety checks would prevent it.

5. Information

--dump-aliases

Instead of the normal operation, dump the shorthand alias names from the configured alias file(s), one per line in alphabetical order. Note that this only includes the alias name and not its expanded email addresses. See *sendemail.aliasesFile* for more information about aliases.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

sendemail.identity

A configuration identity. When given, causes values in the *sendemail.*<*identity*> subsection to take precedence over values in the *sendemail* section. The default identity is the value of *sendemail.identity*.

sendemail.smtpEncryption

See Section G.3.124, “git-send-email(1)” for description. Note that this setting is not subject to the *identity* mechanism.

sendemail.smtpSSLCertPath

Path to ca-certificates (either a directory or a single file). Set it to an empty string to disable certificate verification.

sendemail.<identity>.*

Identity-specific versions of the *sendemail.** parameters found below, taking precedence over those when this identity is selected, through either the command-line or *sendemail.identity*.

sendemail.multiEdit

If true (default), a single editor instance will be spawned to edit files you have to edit (patches when *--annotate* is used, and the summary when *--compose* is used). If false, files will be edited one after the other, spawning a new editor each time.

sendemail.confirm

Sets the default for whether to confirm before sending. Must be one of *always*, *never*, *cc*, *compose*, or *auto*. See *--confirm* in the Section G.3.124, "git-send-email(1)" documentation for the meaning of these values.

sendemail.aliasesFile

To avoid typing long email addresses, point this to one or more email aliases files. You must also supply *sendemail.aliasFileType*.

sendemail.aliasFileType

Format of the file(s) specified in sendemail.aliasesFile. Must be one of *mutt*, *mailrc*, *pine*, *elm*, *gnus*, or *sendmail*.

What an alias file in each format looks like can be found in the documentation of the email program of the same name. The differences and limitations from the standard formats are described below:

sendmail

- Quoted aliases and quoted addresses are not supported: lines that contain a " symbol are ignored.
- Redirection to a file (*/path/name*) or pipe (|*command*) is not supported.
- File inclusion (*:include: /path/name*) is not supported.
- Warnings are printed on the standard error output for any explicitly unsupported constructs, and any other lines that are not recognized by the parser.

sendemail.annotate , sendemail.bcc , sendemail.cc , sendemail.ccCmd , sendemail.chainReplyTo , sendemail.envelopeSender , sendemail.from , sendemail.headerCmd , sendemail.signedOffByCc , sendemail.smtpPass , sendemail.suppressCc , sendemail.suppressFrom , sendemail.to , sendemail.toCmd , sendemail.smtpDomain , sendemail.smtpServer , sendemail.smtpServerPort , sendemail.smtpServerOption , sendemail.smtpUser , sendemail.thread , sendemail.transferEncoding , sendemail.validate , sendemail.xmailer

These configuration variables all provide a default for Section G.3.124, "git-send-email(1)" command-line options. See its documentation for details.

sendemail.signedOffCc (deprecated)

Deprecated alias for *sendemail.signedOffByCc*.

sendemail.smtpBatchSize

Number of messages to be sent per connection, after that a relogin will happen. If the value is 0 or undefined, send all messages in one connection. See also the *--batch-size* option of Section G.3.124, "git-send-email(1)".

sendemail.smtpReloginDelay

Seconds to wait before reconnecting to the smtp server. See also the *--relogin-delay* option of Section G.3.124, “git-send-email(1)”.

sendemail.forbidSendmailVariables

To avoid common misconfiguration mistakes, Section G.3.124, “git-send-email(1)” will abort with a warning if any configuration options for "sendmail" exist. Set this variable to bypass the check.

EXAMPLES

1. Use gmail as the smtp server

To use *git send-email* to send your patches through the GMail SMTP server, edit ~/.gitconfig to specify your account settings:

```
[sendemail]
        smtpEncryption = tls
        smtpServer = smtp.gmail.com
        smtpUser = yourname@gmail.com
        smtpServerPort = 587
```

If you have multi-factor authentication set up on your Gmail account, you can generate an app-specific password for use with *git send-email*. Visit https://security.google.com/settings/security/apppasswords to create it.

Once your commits are ready to be sent to the mailing list, run the following commands:

```
$ git format-patch --cover-letter -M origin/master -o outgoing/
$ edit outgoing/0000-*
$ git send-email outgoing/*
```

The first time you run it, you will be prompted for your credentials. Enter the app-specific or your regular password as appropriate. If you have credential helper configured (see Section G.3.31, “git-credential(1)”), the password will be saved in the credential store so you won't have to type it the next time.

Note: the following core Perl modules that may be installed with your distribution of Perl are required: MIME::Base64, MIME::QuotedPrint, Net::Domain and Net::SMTP. These additional Perl modules are also required: Authen::SASL and Mail::Address.

SEE ALSO

Section G.3.54, “git-format-patch(1)”, Section G.3.68, “git-imap-send(1)”, mbox(5)

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.125. git-send-pack(1)

2024-07-02
Revision History
2024-07-02

NAME

git-send-pack - Push objects over Git protocol to another repository

SYNOPSIS

git send-pack [--mirror] [--dry-run] [--force]
[--receive-pack=<git-receive-pack>]

[--verbose] [--thin] [--atomic]
[--[no-]signed | --signed=(true|false|if-asked)]
[<host>:]<directory> (--all | <ref>…)

DESCRIPTION

Usually you would want to use *git push*, which is a higher-level wrapper of this command, instead. See Section G.3.103, “git-push(1)”.

Invokes *git-receive-pack* on a possibly remote repository, and updates it from the current repository, sending named refs.

OPTIONS

--receive-pack=<git-receive-pack>

Path to the *git-receive-pack* program on the remote end. Sometimes useful when pushing to a remote repository over ssh, and you do not have the program in a directory on the default $PATH.

--exec=<git-receive-pack>

Same as --receive-pack=<git-receive-pack>.

--all

Instead of explicitly specifying which refs to update, update all heads that locally exist.

--stdin

Take the list of refs from stdin, one per line. If there are refs specified on the command line in addition to this option, then the refs from stdin are processed after those on the command line.

If *--stateless-rpc* is specified together with this option then the list of refs must be in packet format (pkt-line). Each ref must be in a separate packet, and the list must end with a flush packet.

--dry-run

Do everything except actually send the updates.

--force

Usually, the command refuses to update a remote ref that is not an ancestor of the local ref used to overwrite it. This flag disables the check. This means that the remote repository can lose commits; use it with care.

--verbose

Run verbosely.

--thin

Send a "thin" pack, which records objects in deltified form based on objects not included in the pack to reduce network traffic.

--atomic

Use an atomic transaction for updating the refs. If any of the refs fails to update then the entire push will fail without changing any refs.

--[no-]signed , --signed=(true|false|if-asked)

GPG-sign the push request to update refs on the receiving side, to allow it to be checked by the hooks and/or be logged. If *false* or *--no-signed*, no signing will be attempted. If *true* or *--signed*, the push will fail if the

server does not support signed pushes. If set to *if-asked*, sign if and only if the server supports signed pushes. The push will also fail if the actual call to *gpg --sign* fails. See Section G.3.108, “git-receive-pack(1)” for the details on the receiving end.

--push-option=<string>

Pass the specified string as a push option for consumption by hooks on the server side. If the server doesn't support push options, error out. See Section G.3.103, “git-push(1)” and Section G.4.7, “githooks(5)” for details.

<host>

A remote host to house the repository. When this part is specified, *git-receive-pack* is invoked via ssh.

<directory>

The repository to update.

<ref>…

The remote refs to update.

SPECIFYING THE REFS

There are three ways to specify which refs to update on the remote end.

With the *--all* flag, all refs that exist locally are transferred to the remote side. You cannot specify any *<ref>* if you use this flag.

Without *--all* and without any *<ref>*, the heads that exist both on the local side and on the remote side are updated.

When one or more *<ref>* are specified explicitly (whether on the command line or via *--stdin*), it can be either a single pattern, or a pair of such patterns separated by a colon ":" (this means that a ref name cannot have a colon in it). A single pattern *<name>* is just shorthand for *<name>:<name>*.

Each pattern pair consists of the source side (before the colon) and the destination side (after the colon). The ref to be pushed is determined by finding a match that matches the source side, and where it is pushed is determined by using the destination side. The rules used to match a ref are the same rules used by *git rev-parse* to resolve a symbolic ref name. See Section G.3.121, “git-rev-parse(1)”.

- It is an error if <src> does not match exactly one of the local refs.
- It is an error if <dst> matches more than one remote ref.
- If <dst> does not match any remote ref, either
  - it has to start with "refs/"; <dst> is used as the destination literally in this case.
  - <src> == <dst> and the ref that matched the <src> must not exist in the set of remote refs; the ref matched <src> locally is used as the name of the destination.

Without *--force*, the <src> ref is stored at the remote only if <dst> does not exist, or <dst> is a proper subset (i.e. an ancestor) of <src>. This check, known as the "fast-forward check", is performed to avoid accidentally overwriting the remote ref and losing other people's commits from there.

With *--force*, the fast-forward check is disabled for all refs.

Optionally, a <ref> parameter can be prefixed with a plus + sign to disable the fast-forward check only on that ref.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.126. git-sh-i18n--envsubst(1)

2024-07-02

Revision History

2024-07-02

NAME

git-sh-i18n--envsubst - Git's own envsubst(1) for i18n fallbacks

SYNOPSIS

```
eval_gettext () {
    printf "%s" "$1" | (
        export PATH $(git sh-i18n--envsubst --variables "$1");
        git sh-i18n--envsubst "$1"
    )
}
```

DESCRIPTION

This is not a command the end user would want to run. Ever. This documentation is meant for people who are studying the plumbing scripts and/or are writing new ones.

git sh-i18n--envsubst is Git's stripped-down copy of the GNU *envsubst(1)* program that comes with the GNU gettext package. It's used internally by Section G.3.127, “git-sh-i18n(1)” to interpolate the variables passed to the *eval_gettext* function.

No promises are made about the interface, or that this program won't disappear without warning in the next version of Git. Don't use it.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.127. git-sh-i18n(1)

2024-07-02

Revision History

2024-07-02

NAME

git-sh-i18n - Git's i18n setup code for shell scripts

SYNOPSIS

```
. "$(git --exec-path)/git-sh-i18n"
```

DESCRIPTION

This is not a command the end user would want to run. Ever. This documentation is meant for people who are studying the Porcelain-ish scripts and/or are writing new ones.

The 'git sh-i18n scriptlet is designed to be sourced (using .) by Git's porcelain programs implemented in shell script. It provides wrappers for the GNU *gettext* and *eval_gettext* functions accessible through the *gettext.sh* script, and provides pass-through fallbacks on systems without GNU gettext.

FUNCTIONS

gettext

Currently a dummy fall-through function implemented as a wrapper around *printf(1)*. Will be replaced by a real gettext implementation in a later version.

eval_gettext

Currently a dummy fall-through function implemented as a wrapper around *printf(1)* with variables expanded by the Section G.3.126, “git-sh-i18n--envsubst(1)” helper. Will be replaced by a real gettext implementation in a later version.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.128. git-sh-setup(1)

2024-07-02

Revision History

2024-07-02

NAME

git-sh-setup - Common Git shell script setup code

SYNOPSIS

. "$(git --exec-path)/git-sh-setup"

DESCRIPTION

This is not a command the end user would want to run. Ever. This documentation is meant for people who are studying the Porcelain-ish scripts and/or are writing new ones.

The *git sh-setup* scriptlet is designed to be sourced (using .) by other shell scripts to set up some variables pointing at the normal Git directories and a few helper shell functions.

Before sourcing it, your script should set up a few variables; *USAGE* (and *LONG_USAGE*, if any) is used to define the message given by *usage()* shell function. *SUBDIRECTORY_OK* can be set if the script can run from a subdirectory of the working tree (some commands do not).

The scriptlet sets *GIT_DIR* and *GIT_OBJECT_DIRECTORY* shell variables, but does **not** export them to the environment.

FUNCTIONS

die

exit after emitting the supplied error message to the standard error stream.

usage

die with the usage message.

set_reflog_action

Set *GIT_REFLOG_ACTION* environment to a given string (typically the name of the program) unless it is already set. Whenever the script runs a *git* command that updates refs, a reflog entry is created using the value of this string to leave the record of what command updated the ref.

git_editor

runs an editor of user's choice (GIT_EDITOR, core.editor, VISUAL or EDITOR) on a given file, but error out if no editor is specified and the terminal is dumb.

is_bare_repository

outputs *true* or *false* to the standard output stream to indicate if the repository is a bare repository (i.e. without an associated working tree).

cd_to_toplevel

runs chdir to the toplevel of the working tree.

require_work_tree

checks if the current directory is within the working tree of the repository, and otherwise dies.

require_work_tree_exists

checks if the working tree associated with the repository exists, and otherwise dies. Often done before calling cd_to_toplevel, which is impossible to do if there is no working tree.

require_clean_work_tree <action> [<hint>]

checks that the working tree and index associated with the repository have no uncommitted changes to tracked files. Otherwise it emits an error message of the form *Cannot <action>: <reason>. <hint>*, and dies. Example:

```
require_clean_work_tree rebase "Please commit or stash them."
```

get_author_ident_from_commit

outputs code for use with eval to set the GIT_AUTHOR_NAME, GIT_AUTHOR_EMAIL and GIT_AUTHOR_DATE variables for a given commit.

create_virtual_base

modifies the first file so only lines in common with the second file remain. If there is insufficient common material, then the first file is left empty. The result is suitable as a virtual base input for a 3-way merge.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.129. git-shell(1)

2024-07-02
Revision History
2024-07-02

NAME

git-shell - Restricted login shell for Git-only SSH access

SYNOPSIS

chsh -s $(command -v git-shell) <user>
git clone <user>*@localhost:/path/to/repo.git*
ssh <user>*@localhost*

DESCRIPTION

This is a login shell for SSH accounts to provide restricted Git access. It permits execution only of server-side Git commands implementing the pull/push functionality, plus custom commands present in a subdirectory named *git-shell-commands* in the user's home directory.

COMMANDS

git shell accepts the following commands after the *-c* option:

git receive-pack <argument> , *git upload-pack <argument>* , *git upload-archive <argument>*

Call the corresponding server-side command to support the client's *git push*, *git fetch*, or *git archive --remote* request.

cvs server

Imitate a CVS server. See Section G.3.37, “git-cvsserver(1)”.

If a *~/git-shell-commands* directory is present, *git shell* will also handle other, custom commands by running "*git-shell-commands/<command> <arguments>*" from the user's home directory.

INTERACTIVE USE

By default, the commands above can be executed only with the *-c* option; the shell is not interactive.

If a *~/git-shell-commands* directory is present, *git shell* can also be run interactively (with no arguments). If a *help* command is present in the *git-shell-commands* directory, it is run to provide the user with an overview of allowed actions. Then a "git> " prompt is presented at which one can enter any of the commands from the *git-shell-commands* directory, or *exit* to close the connection.

Generally this mode is used as an administrative interface to allow users to list repositories they have access to, create, delete, or rename repositories, or change repository descriptions and permissions.

If a *no-interactive-login* command exists, then it is run and the interactive shell is aborted.

EXAMPLES

To disable interactive logins, displaying a greeting instead:

```
$ chsh -s /usr/bin/git-shell
$ mkdir $HOME/git-shell-commands
$ cat >$HOME/git-shell-commands/no-interactive-login <<\EOF
#!/bin/sh
printf '%s\n' "Hi $USER! You've successfully authenticated, but I do not"
printf '%s\n' "provide interactive shell access."
exit 128
EOF
$ chmod +x $HOME/git-shell-commands/no-interactive-login
```

To enable git-cvsserver access (which should generally have the *no-interactive-login* example above as a prerequisite, as creating the git-shell-commands directory allows interactive logins):

```
$ cat >$HOME/git-shell-commands/cvs <<\EOF
if ! test $# = 1 && test "$1" = "server"
then
        echo >&2 "git-cvsserver only handles \"server\""
        exit 1
fi
exec git cvsserver server
EOF
$ chmod +x $HOME/git-shell-commands/cvs
```

SEE ALSO

ssh(1), Section G.3.38, “git-daemon(1)”, contrib/git-shell-commands/README

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.130. git-shortlog(1)

2024-07-02

Revision History

2024-07-02

NAME

git-shortlog - Summarize *git log* output

SYNOPSIS

git shortlog [<options>] [<revision-range>] [[--] <path>…]
git log --pretty=short | *git shortlog* [<options>]

DESCRIPTION

Summarizes *git log* output in a format suitable for inclusion in release announcements. Each commit will be grouped by author and title.

Additionally, "[PATCH]" will be stripped from the commit description.

If no revisions are passed on the command line and either standard input is not a terminal or there is no current branch, *git shortlog* will output a summary of the log read from standard input, without reference to the current repository.

OPTIONS

-n , --numbered

Sort output according to the number of commits per author instead of author alphabetic order.

-s , --summary

Suppress commit description and provide a commit count summary only.

-e , --email

Show the email address of each author.

--format[=<format>]

Instead of the commit subject, use some other information to describe each commit. <*format*> can be any string accepted by the --*format* option of *git log*, such as * *[%h] %s*. (See the "PRETTY FORMATS" section of Section G.3.74, “git-log(1)”.)

```
Each pretty-printed commit will be rewrapped before it is shown.
```

--date=<format>

Show dates formatted according to the given date string. (See the --*date* option in the "Commit Formatting" section of Section G.3.74, “git-log(1)”). Useful with --*group=format:*<*format*>.

--group=<type>

Group commits based on <*type*>. If no --*group* option is specified, the default is *author*. <*type*> is one of:

- *author*, commits are grouped by author
- *committer*, commits are grouped by committer (the same as -*c*)
- *trailer:*<*field*>, the <*field*> is interpreted as a case-insensitive commit message trailer (see Section G.3.73, “git-interpret-trailers(1)”). For example, if your project uses *Reviewed-by* trailers, you might want to see who has been reviewing with *git shortlog -ns --group=trailer:reviewed-by*.
- *format:*<*format*>, any string accepted by the --*format* option of *git log*. (See the "PRETTY FORMATS" section of Section G.3.74, “git-log(1)”.)

Note that commits that do not include the trailer will not be counted. Likewise, commits with multiple trailers (e.g., multiple signoffs) may be counted more than once (but only once per unique trailer value in that commit).

Shortlog will attempt to parse each trailer value as a *name* <*email*> identity. If successful, the mailmap is applied and the email is omitted unless the *--email* option is specified. If the value cannot be parsed as an identity, it will be taken literally and completely.

If *--group* is specified multiple times, commits are counted under each value (but again, only once per unique value in that commit). For example, *git shortlog --group=author --group=trailer:co-authored-by* counts both authors and co-authors.

-c , --committer

This is an alias for *--group=committer*.

-w[<width>[,<indent1>[,<indent2>]]]

Linewrap the output by wrapping each line at *width*. The first line of each entry is indented by *indent1* spaces, and the second and subsequent lines are indented by *indent2* spaces. *width*, *indent1*, and *indent2* default to 76, 6 and 9 respectively.

If width is *0* (zero) then indent the lines of the output without wrapping them.

<revision-range>

Show only commits in the specified revision range. When no <revision-range> is specified, it defaults to *HEAD* (i.e. the whole history leading to the current commit). *origin..HEAD* specifies all the commits reachable from the current commit (i.e. *HEAD*), but not from *origin*. For a complete list of ways to spell <revision-range>, see the "Specifying Ranges" section of Section G.4.14, “gitrevisions(7)”.

[--] <path>…

Consider only commits that are enough to explain how the files that match the specified paths came to be.

Paths may need to be prefixed with -- to separate them from options or the revision range, when confusion arises.

1. Commit Limiting

Besides specifying a range of commits that should be listed using the special notations explained in the description, additional commit limiting may be applied.

Using more options generally further limits the output (e.g. *--since=<date1>* limits to commits newer than *<date1>*, and using it with *--grep=<pattern>* further limits to commits whose log message has a line that matches *<pattern>*), unless otherwise noted.

Note that these are applied before commit ordering and formatting options, such as *--reverse*.

-<number> , -n <number> , --max-count=<number>

Limit the number of commits to output.

--skip=<number>

Skip *number* commits before starting to show the commit output.

--since=<date> , --after=<date>

Show commits more recent than a specific date.

--since-as-filter=<date>

Show all commits more recent than a specific date. This visits all commits in the range, rather than stopping at the first commit which is older than a specific date.

--until=<date> , --before=<date>

Show commits older than a specific date.

--author=<pattern> , --committer=<pattern>

Limit the commits output to ones with author/committer header lines that match the specified pattern (regular expression). With more than one *--author=<pattern>*, commits whose author matches any of the given patterns are chosen (similarly for multiple *--committer=<pattern>*).

--grep-reflog=<pattern>

Limit the commits output to ones with reflog entries that match the specified pattern (regular expression). With more than one *--grep-reflog*, commits whose reflog message matches any of the given patterns are chosen. It is an error to use this option unless *--walk-reflogs* is in use.

--grep=<pattern>

Limit the commits output to ones with a log message that matches the specified pattern (regular expression). With more than one *--grep=<pattern>*, commits whose message matches any of the given patterns are chosen (but see *--all-match*).

When *--notes* is in effect, the message from the notes is matched as if it were part of the log message.

--all-match

Limit the commits output to ones that match all given *--grep*, instead of ones that match at least one.

--invert-grep

Limit the commits output to ones with a log message that do not match the pattern specified with *--grep=<pattern>*.

-i , --regexp-ignore-case

Match the regular expression limiting patterns without regard to letter case.

--basic-regexp

Consider the limiting patterns to be basic regular expressions; this is the default.

-E , --extended-regexp

Consider the limiting patterns to be extended regular expressions instead of the default basic regular expressions.

-F , --fixed-strings

Consider the limiting patterns to be fixed strings (don't interpret pattern as a regular expression).

-P , --perl-regexp

Consider the limiting patterns to be Perl-compatible regular expressions.

Support for these types of regular expressions is an optional compile-time dependency. If Git wasn't compiled with support for them providing this option will cause it to die.

--remove-empty

Stop when a given path disappears from the tree.

--merges

Print only merge commits. This is exactly the same as *--min-parents=2*.

--no-merges

Do not print commits with more than one parent. This is exactly the same as *--max-parents=1*.

--min-parents=<number> , --max-parents=<number> , --no-min-parents , --no-max-parents

Show only commits which have at least (or at most) that many parent commits. In particular, *--max-parents=1* is the same as *--no-merges*, *--min-parents=2* is the same as *--merges*. *--max-parents=0* gives all root commits and *--min-parents=3* all octopus merges.

--no-min-parents and *--no-max-parents* reset these limits (to no limit) again. Equivalent forms are *--min-parents=0* (any commit has 0 or more parents) and *--max-parents=-1* (negative numbers denote no upper limit).

--first-parent

When finding commits to include, follow only the first parent commit upon seeing a merge commit. This option can give a better overview when viewing the evolution of a particular topic branch, because merges into a topic branch tend to be only about adjusting to updated upstream from time to time, and this option allows you to ignore the individual commits brought in to your history by such a merge.

--exclude-first-parent-only

When finding commits to exclude (with a ^), follow only the first parent commit upon seeing a merge commit. This can be used to find the set of changes in a topic branch from the point where it diverged from the remote branch, given that arbitrary merges can be valid topic branch changes.

--not

Reverses the meaning of the ^ prefix (or lack thereof) for all following revision specifiers, up to the next *--not*. When used on the command line before --stdin, the revisions passed through stdin will not be affected by it. Conversely, when passed via standard input, the revisions passed on the command line will not be affected by it.

--all

Pretend as if all the refs in *refs/*, along with *HEAD*, are listed on the command line as *<commit>*.

--branches[=<pattern>]

Pretend as if all the refs in *refs/heads* are listed on the command line as *<commit>*. If *<pattern>* is given, limit branches to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--tags[=<pattern>]

Pretend as if all the refs in *refs/tags* are listed on the command line as *<commit>*. If *<pattern>* is given, limit tags to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--remotes[=<pattern>]

Pretend as if all the refs in *refs/remotes* are listed on the command line as *<commit>*. If *<pattern>* is given, limit remote-tracking branches to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--glob=<glob-pattern>

Pretend as if all the refs matching shell glob *<glob-pattern>* are listed on the command line as *<commit>*. Leading *refs/*, is automatically prepended if missing. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--exclude=<glob-pattern>

Do not include refs matching *<glob-pattern>* that the next *--all*, *--branches*, *--tags*, *--remotes*, or *--glob* would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next *--all*, *--branches*, *--tags*, *--remotes*, or *--glob* option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with *refs/heads*, *refs/tags*, or *refs/remotes* when applied to *--branches*, *--tags*, or *--remotes*, respectively, and they must begin with *refs/* when applied to *--glob* or *--all*. If a trailing /* is intended, it must be given explicitly.

--exclude-hidden=[fetch|receive|uploadpack]

Do not include refs that would be hidden by *git-fetch*, *git-receive-pack* or *git-upload-pack* by consulting the appropriate *fetch.hideRefs*, *receive.hideRefs* or *uploadpack.hideRefs* configuration along with *transfer.hideRefs* (see Section G.3.29, "git-config(1)"). This option affects the next pseudo-ref option *--all* or *--glob* and is cleared after processing them.

--reflog

Pretend as if all objects mentioned by reflogs are listed on the command line as *<commit>*.

--alternate-refs

Pretend as if all objects mentioned as ref tips of alternate repositories were listed on the command line. An alternate repository is any repository whose object directory is specified in *objects/info/alternates*. The set of included objects may be modified by *core.alternateRefsCommand*, etc. See Section G.3.29, "git-config(1)".

--single-worktree

By default, all working trees will be examined by the following options when there are more than one (see Section G.3.159, "git-worktree(1)"): *--all*, *--reflog* and *--indexed-objects*. This option forces them to examine the current working tree only.

--ignore-missing

Upon seeing an invalid object name in the input, pretend as if the bad input was not given.

--bisect

Pretend as if the bad bisection ref *refs/bisect/bad* was listed and as if it was followed by *--not* and the good bisection refs *refs/bisect/good-** on the command line.

--stdin

In addition to getting arguments from the command line, read them from standard input as well. This accepts commits and pseudo-options like *--all* and *--glob=*. When a -- separator is seen, the following input is treated as paths and used to limit the result. Flags like *--not* which are read via standard input are only respected for arguments passed in the same way and will not influence any subsequent command line arguments.

--cherry-mark

Like *--cherry-pick* (see below) but mark equivalent commits with = rather than omitting them, and inequivalent ones with +.

--cherry-pick

Omit any commit that introduces the same change as another commit on the other side when the set of commits are limited with symmetric difference.

For example, if you have two branches, *A* and *B*, a usual way to list all commits on only one side of them is with *--left-right* (see the example below in the description of the *--left-right* option). However, it shows the commits that were cherry-picked from the other branch (for example, 3rd on b may be cherry-picked from branch A). With this option, such pairs of commits are excluded from the output.

--left-only , --right-only

List only commits on the respective side of a symmetric difference, i.e. only those which would be marked < resp. > by *--left-right*.

For example, *--cherry-pick --right-only A...B* omits those commits from *B* which are in *A* or are patch-equivalent to a commit in *A*. In other words, this lists the + commits from *git cherry A B*. More precisely, *--cherry-pick --right-only --no-merges* gives the exact list.

--cherry

A synonym for *--right-only --cherry-mark --no-merges*; useful to limit the output to the commits on our side and mark those that have been applied to the other side of a forked history with *git log --cherry upstream...mybranch*, similar to *git cherry upstream mybranch*.

-g , --walk-reflogs

Instead of walking the commit ancestry chain, walk reflog entries from the most recent one to older ones. When this option is used you cannot specify commits to exclude (that is, *^commit*, *commit1..commit2*, and *commit1...commit2* notations cannot be used).

With *--pretty* format other than *oneline* and *reference* (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. The reflog designator in the output may be shown as *ref@{<Nth>}* (where *<Nth>* is the reverse-chronological index in the reflog) or as *ref@{<timestamp>}* (with the *<timestamp>* for that entry), depending on a few rules:

1. If the starting point is specified as *ref@{<Nth>}*, show the index format.
2. If the starting point was specified as *ref@{now}*, show the timestamp format.
3. If neither was used, but *--date* was given on the command line, show the timestamp in the format requested by *--date*.
4. Otherwise, show the index format.

Under *--pretty=oneline*, the commit message is prefixed with this information on the same line. This option cannot be combined with *--reverse*. See also Section G.3.109, “git-reflog(1)”.

Under *--pretty=reference*, this information will not be shown at all.

--merge

Show commits touching conflicted paths in the range *HEAD...<other>*, where *<other>* is the first existing pseudoref in *MERGE_HEAD*, *CHERRY_PICK_HEAD*, *REVERT_HEAD* or *REBASE_HEAD*. Only works when the index has unmerged entries. This option can be used to show relevant commits when resolving conflicts from a 3-way merge.

--boundary

Output excluded boundary commits. Boundary commits are prefixed with -.

2. History Simplification

Sometimes you are only interested in parts of the history, for example the commits modifying a particular <path>. But there are two parts of *History Simplification*, one part is selecting the commits and the other is how to do it, as there are various strategies to simplify the history.

The following options select the commits to be shown:

<paths>

Commits modifying the given <paths> are selected.

--simplify-by-decoration

Commits that are referred by some branch or tag are selected.

Note that extra commits can be shown to give a meaningful history.

The following options affect the way the simplification is performed:

Default mode

Simplifies the history to the simplest history explaining the final state of the tree. Simplest because it prunes some side branches if the end result is the same (i.e. merging branches with the same content)

--show-pulls

Include all commits from the default mode, but also any merge commits that are not TREESAME to the first parent but are TREESAME to a later parent. This mode is helpful for showing the merge commits that "first introduced" a change to a branch.

--full-history

Same as the default mode, but does not prune some history.

--dense

Only the selected commits are shown, plus some to have a meaningful history.

--sparse

All commits in the simplified history are shown.

--simplify-merges

Additional option to *--full-history* to remove some needless merges from the resulting history, as there are no selected commits contributing to this merge.

--ancestry-path[=<commit>]

When given a range of commits to display (e.g. *commit1..commit2* or *commit2 ^commit1*), only display commits in that range that are ancestors of <commit>, descendants of <commit>, or <commit> itself. If no commit is specified, use *commit1* (the excluded part of the range) as <commit>. Can be passed multiple times; if so, a commit is included if it is any of the commits given or if it is an ancestor or descendant of one of them.

A more detailed explanation follows.

Suppose you specified *foo* as the <paths>. We shall call commits that modify *foo* !TREESAME, and the rest TREESAME. (In a diff filtered for *foo*, they look different and equal, respectively.)

In the following, we will always refer to the same example history to illustrate the differences between simplification settings. We assume that you are filtering for a file *foo* in this commit graph:

```
     .-A---M---N---O---P---Q
    /     /   /   /   /   /
   I     B   C   D   E   Y
    \   /   /   /   /   /
     `-------------'   X
```

The horizontal line of history A---Q is taken to be the first parent of each merge. The commits are:

- *I* is the initial commit, in which *foo* exists with contents asdf, and a file *quux* exists with contents quux. Initial commits are compared to an empty tree, so *I* is !TREESAME.
- In *A*, *foo* contains just foo.

- *B* contains the same change as *A*. Its merge *M* is trivial and hence TREESAME to all parents.
- *C* does not change *foo*, but its merge *N* changes it to foobar, so it is not TREESAME to any parent.
- *D* sets *foo* to baz. Its merge *O* combines the strings from *N* and *D* to foobarbaz; i.e., it is not TREESAME to any parent.
- *E* changes *quux* to xyzzy, and its merge *P* combines the strings to quux xyzzy. *P* is TREESAME to *O*, but not to *E*.
- *X* is an independent root commit that added a new file *side*, and *Y* modified it. *Y* is TREESAME to *X*. Its merge *Q* added *side* to *P*, and *Q* is TREESAME to *P*, but not to *Y*.

rev-list walks backwards through history, including or excluding commits based on whether --*full-history* and/or parent rewriting (via --*parents* or --*children*) are used. The following settings are available.

Default mode

Commits are included if they are not TREESAME to any parent (though this can be changed, see --*sparse* below). If the commit was a merge, and it was TREESAME to one parent, follow only that parent. (Even if there are several TREESAME parents, follow only one of them.) Otherwise, follow all parents.

This results in:

```
  .-A---N---O
 /     /   /
I---------D
```

Note how the rule to only follow the TREESAME parent, if one is available, removed *B* from consideration entirely. *C* was considered via *N*, but is TREESAME. Root commits are compared to an empty tree, so *I* is !TREESAME.

Parent/child relations are only visible with --*parents*, but that does not affect the commits selected in default mode, so we have shown the parent lines.

--full-history without parent rewriting

This mode differs from the default in one point: always follow all parents of a merge, even if it is TREESAME to one of them. Even if more than one side of the merge has commits that are included, this does not imply that the merge itself is! In the example, we get

```
I  A  B  N  D  O  P  Q
```

M was excluded because it is TREESAME to both parents. *E*, *C* and *B* were all walked, but only *B* was !TREESAME, so the others do not appear.

Note that without parent rewriting, it is not really possible to talk about the parent/child relationships between the commits, so we show them disconnected.

--full-history with parent rewriting

Ordinary commits are only included if they are !TREESAME (though this can be changed, see --*sparse* below).

Merges are always included. However, their parent list is rewritten: Along each parent, prune away commits that are not included themselves. This results in

```
  .-A---M---N---O---P---Q
 /     /   /   /   /   /
I     B   /   D   /
 \   /   /   /   /
```

```
                `-------------'
```

Compare to *--full-history* without rewriting above. Note that *E* was pruned away because it is TREESAME, but the parent list of P was rewritten to contain *E*'s parent *I*. The same happened for *C* and *N*, and *X*, *Y* and *Q*.

In addition to the above settings, you can change whether TREESAME affects inclusion:

--dense

Commits that are walked are included if they are not TREESAME to any parent.

--sparse

All commits that are walked are included.

Note that without *--full-history*, this still simplifies merges: if one of the parents is TREESAME, we follow only that one, so the other sides of the merge are never walked.

--simplify-merges

First, build a history graph in the same way that *--full-history* with parent rewriting does (see above).

Then simplify each commit *C* to its replacement *C'* in the final history according to the following rules:

- Set *C'* to *C*.
- Replace each parent *P* of *C'* with its simplification *P'*. In the process, drop parents that are ancestors of other parents or that are root commits TREESAME to an empty tree, and remove duplicates, but take care to never drop all parents that we are TREESAME to.
- If after this parent rewriting, *C'* is a root or merge commit (has zero or >1 parents), a boundary commit, or !TREESAME, it remains. Otherwise, it is replaced with its only parent.

The effect of this is best shown by way of comparing to *--full-history* with parent rewriting. The example turns into:

```
  .-A---M---N---O
 /     /       /
I     B       D
 \   /       /
  `---------'
```

Note the major differences in *N*, *P*, and *Q* over *--full-history*:

- *N*'s parent list had *I* removed, because it is an ancestor of the other parent *M*. Still, *N* remained because it is !TREESAME.
- *P*'s parent list similarly had *I* removed. *P* was then removed completely, because it had one parent and is TREESAME.
- *Q*'s parent list had *Y* simplified to *X*. *X* was then removed, because it was a TREESAME root. *Q* was then removed completely, because it had one parent and is TREESAME.

There is another simplification mode available:

--ancestry-path[=<commit>]

Limit the displayed commits to those which are an ancestor of <commit>, or which are a descendant of <commit>, or are <commit> itself.

As an example use case, consider the following commit history:

```
          D---E-------F
```

```
          /     \       \
         B---C---G---H---I---J
        /                     \
       A-------K---------------L--M
```

A regular *D..M* computes the set of commits that are ancestors of *M*, but excludes the ones that are ancestors of *D*. This is useful to see what happened to the history leading to *M* since *D*, in the sense that what does *M* have that did not exist in *D*. The result in this example would be all the commits, except *A* and *B* (and *D* itself, of course).

When we want to find out what commits in *M* are contaminated with the bug introduced by *D* and need fixing, however, we might want to view only the subset of *D..M* that are actually descendants of *D*, i.e. excluding *C* and *K*. This is exactly what the *--ancestry-path* option does. Applied to the *D..M* range, it results in:

```
E-------F
 \       \
  G---H---I---J
                \
                 L--M
```

We can also use *--ancestry-path=D* instead of *--ancestry-path* which means the same thing when applied to the *D..M* range but is just more explicit.

If we instead are interested in a given topic within this range, and all commits affected by that topic, we may only want to view the subset of *D..M* which contain that topic in their ancestry path. So, using *--ancestry-path=H D..M* for example would result in:

```
E
 \
  G---H---I---J
                \
                 L--M
```

Whereas *--ancestry-path=K D..M* would result in

```
K---------------L--M
```

Before discussing another option, *--show-pulls*, we need to create a new example history.

A common problem users face when looking at simplified history is that a commit they know changed a file somehow does not appear in the file's simplified history. Let's demonstrate a new example and show how options such as *--full-history* and *--simplify-merges* works in that case:

```
  .-A---M-----C--N---O---P
 /     / \  \  \/   /   /
I     B   \  R-'`-Z'   /
 \   /     \/         /
  \ /      /\        /
   `---X--'   `---Y--'
```

For this example, suppose *I* created *file.txt* which was modified by *A*, *B*, and *X* in different ways. The single-parent commits *C*, *Z*, and *Y* do not change *file.txt*. The merge commit *M* was created by resolving the merge conflict to include both changes from *A* and *B* and hence is not TREESAME to either. The merge commit *R*, however, was created by ignoring the contents of *file.txt* at *M* and taking only the contents of *file.txt* at *X*. Hence, *R* is TREESAME to *X* but not *M*. Finally, the natural merge resolution to create *N* is to take the contents of *file.txt* at *R*, so *N* is TREESAME to *R* but not *C*. The merge commits *O* and *P* are TREESAME to their first parents, but not to their second parents, *Z* and *Y* respectively.

When using the default mode, *N* and *R* both have a TREESAME parent, so those edges are walked and the others are ignored. The resulting history graph is:

```
I---X
```

When using *--full-history*, Git walks every edge. This will discover the commits *A* and *B* and the merge *M*, but also will reveal the merge commits *O* and *P*. With parent rewriting, the resulting graph is:

```
  .-A---M--------N---O---P
 /     / \  \  \/   /   /
I     B   \  R-'`--'   /
 \   /     \/         /
  \ /      /\        /
   `---X--'   `------'
```

Here, the merge commits *O* and *P* contribute extra noise, as they did not actually contribute a change to *file.txt*. They only merged a topic that was based on an older version of *file.txt*. This is a common issue in repositories using a workflow where many contributors work in parallel and merge their topic branches along a single trunk: many unrelated merges appear in the *--full-history* results.

When using the *--simplify-merges* option, the commits *O* and *P* disappear from the results. This is because the rewritten second parents of *O* and *P* are reachable from their first parents. Those edges are removed and then the commits look like single-parent commits that are TREESAME to their parent. This also happens to the commit *N*, resulting in a history view as follows:

```
  .-A---M--.
 /     /    \
I     B      R
 \   /      /
  \ /      /
   `---X--'
```

In this view, we see all of the important single-parent changes from *A*, *B*, and *X*. We also see the carefully-resolved merge *M* and the not-so-carefully-resolved merge *R*. This is usually enough information to determine why the commits *A* and *B* "disappeared" from history in the default view. However, there are a few issues with this approach.

The first issue is performance. Unlike any previous option, the *--simplify-merges* option requires walking the entire commit history before returning a single result. This can make the option difficult to use for very large repositories.

The second issue is one of auditing. When many contributors are working on the same repository, it is important which merge commits introduced a change into an important branch. The problematic merge *R* above is not likely to be the merge commit that was used to merge into an important branch. Instead, the merge *N* was used to merge *R* and *X* into the important branch. This commit may have information about why the change *X* came to override the changes from *A* and *B* in its commit message.

--show-pulls

In addition to the commits shown in the default history, show each merge commit that is not TREESAME to its first parent but is TREESAME to a later parent.

When a merge commit is included by *--show-pulls*, the merge is treated as if it "pulled" the change from another branch. When using *--show-pulls* on this example (and no other options) the resulting graph is:

```
I---X---R---N
```

Here, the merge commits *R* and *N* are included because they pulled the commits *X* and *R* into the base branch, respectively. These merges are the reason the commits *A* and *B* do not appear in the default history.

When *--show-pulls* is paired with *--simplify-merges*, the graph includes all of the necessary information:

```
  .-A---M--.   N
 /     /    \ /
I     B      R
```

```
\   /       /
 \ /       /
  `---X--'
```

Notice that since *M* is reachable from *R*, the edge from *N* to *M* was simplified away. However, *N* still appears in the history as an important commit because it "pulled" the change *R* into the main branch.

The *--simplify-by-decoration* option allows you to view only the big picture of the topology of the history, by omitting commits that are not referenced by tags. Commits are marked as !TREESAME (in other words, kept after history simplification rules described above) if (1) they are referenced by tags, or (2) they change the contents of the paths given on the command line. All other commits are marked as TREESAME (subject to be simplified away).

MAPPING AUTHORS

See Section G.4.9, “gitmailmap(5)”.

Note that if *git shortlog* is run outside of a repository (to process log contents on standard input), it will look for a *.mailmap* file in the current directory.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.131. git-show-branch(1)

2024-07-02
Revision History
2024-07-02

NAME

git-show-branch - Show branches and their commits

SYNOPSIS

git show-branch [-a | --all] [-r | --remotes] [--topo-order | --date-order]
[--current] [--color[=<when>] | --no-color] [--sparse]
[--more=<n> | --list | --independent | --merge-base]
[--no-name | --sha1-name] [--topics]
[(<rev> | <glob>)…]
git show-branch (-g | --reflog)[=<n>[,<base>]] [--list] [<ref>]

DESCRIPTION

Shows the commit ancestry graph starting from the commits named with <rev>s or <glob>s (or all refs under refs/heads and/or refs/tags) semi-visually.

It cannot show more than 29 branches and commits at a time.

It uses *showbranch.default* multi-valued configuration items if no <rev> or <glob> is given on the command line.

OPTIONS

<rev>

Arbitrary extended SHA-1 expression (see Section G.4.14, “gitrevisions(7)”) that typically names a branch head or a tag.

<glob>

A glob pattern that matches branch or tag names under refs/. For example, if you have many topic branches under refs/heads/topic, giving *topic/** would show all of them.

-r , --remotes

Show the remote-tracking branches.

-a , --all

Show both remote-tracking branches and local branches.

--current

With this option, the command includes the current branch in the list of revs to be shown when it is not given on the command line.

--topo-order

By default, the branches and their commits are shown in reverse chronological order. This option makes them appear in topological order (i.e., descendant commits are shown before their parents).

--date-order

This option is similar to *--topo-order* in the sense that no parent comes before all of its children, but otherwise commits are ordered according to their commit date.

--sparse

By default, the output omits merges that are reachable from only one tip being shown. This option makes them visible.

--more=<n>

Usually the command stops output upon showing the commit that is the common ancestor of all the branches. This flag tells the command to go <n> more common commits beyond that. When <n> is negative, display only the <ref>s given, without showing the commit ancestry tree.

--list

Synonym to *--more=-1*

--merge-base

Instead of showing the commit list, determine possible merge bases for the specified commits. All merge bases will be contained in all specified commits. This is different from how Section G.3.81, “git-merge-base(1)” handles the case of three or more commits.

--independent

Among the <ref>s given, display only the ones that cannot be reached from any other <ref>.

--no-name

Do not show naming strings for each commit.

--sha1-name

Instead of naming the commits using the path to reach them from heads (e.g. "master~2" to mean the grandparent of "master"), name them with the unique prefix of their object names.

--topics

Shows only commits that are NOT on the first branch given. This helps track topic branches by hiding any commit that is already in the main line of development. When given "git show-branch --topics master topic1 topic2", this will show the revisions given by "git rev-list ^master topic1 topic2"

-g , --reflog[=<n>[,<base>]] [<ref>]

Shows <n> most recent ref-log entries for the given ref. If <base> is given, <n> entries going back from that entry. <base> can be specified as count or date. When no explicit <ref> parameter is given, it defaults to the current branch (or *HEAD* if it is detached).

--color[=<when>]

Color the status sign (one of these: * *!* + -) of each commit corresponding to the branch it's in. The value must be always (the default), never, or auto.

--no-color

Turn off colored output, even when the configuration file gives the default to color output. Same as *--color=never*.

Note that --more, --list, --independent, and --merge-base options are mutually exclusive.

OUTPUT

Given N <ref>s, the first N lines are the one-line description from their commit message. The branch head that is pointed at by $GIT_DIR/HEAD is prefixed with an asterisk * character while other heads are prefixed with a *!* character.

Following these N lines, a one-line log for each commit is displayed, indented N places. If a commit is on the I-th branch, the I-th indentation character shows a + sign; otherwise it shows a space. Merge commits are denoted by a - sign. Each commit shows a short name that can be used as an extended SHA-1 to name that commit.

The following example shows three branches, "master", "fixes", and "mhf":

```
$ git show-branch master fixes mhf
* [master] Add 'git show-branch'.
 ! [fixes] Introduce "reset type" flag to "git reset"
  ! [mhf] Allow "+remote:local" refspec to cause --force when fetching.
---
  + [mhf] Allow "+remote:local" refspec to cause --force when fetching.
  + [mhf~1] Use git-octopus when pulling more than one head.
 +  [fixes] Introduce "reset type" flag to "git reset"
  + [mhf~2] "git fetch --force".
  + [mhf~3] Use .git/remote/origin, not .git/branches/origin.
  + [mhf~4] Make "git pull" and "git fetch" default to origin
  + [mhf~5] Infamous 'octopus merge'
  + [mhf~6] Retire git-parse-remote.
  + [mhf~7] Multi-head fetch.
  + [mhf~8] Start adding the $GIT_DIR/remotes/ support.
*++ [master] Add 'git show-branch'.
```

These three branches all forked from a common commit, [master], whose commit message is "Add 'git show-branch'". The "fixes" branch adds one commit "Introduce "reset type" flag to "git reset"". The "mhf" branch adds many other commits. The current branch is "master".

EXAMPLES

If you keep your primary branches immediately under *refs/heads*, and topic branches in subdirectories of it, having the following in the configuration file may help:

```
[showbranch]
        default = --topo-order
        default = heads/*
```

With this, *git show-branch* without extra parameters would show only the primary branches. In addition, if you happen to be on your topic branch, it is shown as well.

```
$ git show-branch --reflog="10,1 hour ago" --list master
```

shows 10 reflog entries going back from the tip as of 1 hour ago. Without *--list*, the output also shows how these tips are topologically related to each other.

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, "git-config(1)" documentation. The content is the same as what's found there:

showBranch.default

The default set of branches for Section G.3.131, "git-show-branch(1)". See Section G.3.131, "git-show-branch(1)".

GIT

Part of the Section G.3.1, "git(1)" suite

G.3.132. git-show-index(1)

2024-07-02
Revision History
2024-07-02

NAME

git-show-index - Show packed archive index

SYNOPSIS

git show-index [--object-format=<hash-algorithm>]

DESCRIPTION

Read the *.idx* file for a Git packfile (created with Section G.3.96, "git-pack-objects(1)" or Section G.3.69, "git-index-pack(1)") from the standard input, and dump its contents. The output consists of one object per line, with each line containing two or three space-separated columns:

- the first column is the offset in bytes of the object within the corresponding packfile
- the second column is the object id of the object
- if the index version is 2 or higher, the third column contains the CRC32 of the object data

The objects are output in the order in which they are found in the index file, which should be (in a correctly constructed file) sorted by object id.

Note that you can get more information on a packfile by calling Section G.3.154, "git-verify-pack(1)". However, as this command considers only the index file itself, it's both faster and more flexible.

OPTIONS

--object-format=<hash-algorithm>

Specify the given object format (hash algorithm) for the index file. The valid values are *sha1* and (if enabled) *sha256*. The default is the algorithm for the current repository (set by *extensions.objectFormat*), or *sha1* if no value is set or outside a repository..

Note: At present, there is no interoperability between SHA-256 repositories and SHA-1 repositories.

Historically, we warned that SHA-256 repositories may later need backward incompatible changes when we introduce such interoperability features. Today, we only expect compatible changes. Furthermore, if such changes

prove to be necessary, it can be expected that SHA-256 repositories created with today's Git will be usable by future versions of Git without data loss.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.133. git-show-ref(1)

2024-07-02
Revision History
2024-07-02

NAME

git-show-ref - List references in a local repository

SYNOPSIS

git show-ref [--head] [-d | --dereference]
[-s | --hash[=<n>]] [--abbrev[=<n>]] [--tags]
[--heads] [--] [<pattern>…]
git show-ref --verify [-q | --quiet] [-d | --dereference]
[-s | --hash[=<n>]] [--abbrev[=<n>]]
[--] [<ref>…]
git show-ref --exclude-existing[=<pattern>]
git show-ref --exists <ref>

DESCRIPTION

Displays references available in a local repository along with the associated commit IDs. Results can be filtered using a pattern and tags can be dereferenced into object IDs. Additionally, it can be used to test whether a particular ref exists.

By default, shows the tags, heads, and remote refs.

The *--exclude-existing* form is a filter that does the inverse. It reads refs from stdin, one ref per line, and shows those that don't exist in the local repository.

The *--exists* form can be used to check for the existence of a single references. This form does not verify whether the reference resolves to an actual object.

Use of this utility is encouraged in favor of directly accessing files under the *.git* directory.

OPTIONS

--head

Show the HEAD reference, even if it would normally be filtered out.

--heads , --tags

Limit to "refs/heads" and "refs/tags", respectively. These options are not mutually exclusive; when given both, references stored in "refs/heads" and "refs/tags" are displayed.

-d , --dereference

Dereference tags into object IDs as well. They will be shown with *^{}* appended.

-s , --hash[=<n>]

Only show the OID, not the reference name. When combined with *--dereference*, the dereferenced tag will still be shown after the OID.

--verify

Enable stricter reference checking by requiring an exact ref path. Aside from returning an error code of 1, it will also print an error message if *--quiet* was not specified.

--exists

Check whether the given reference exists. Returns an exit code of 0 if it does, 2 if it is missing, and 1 in case looking up the reference failed with an error other than the reference being missing.

--abbrev[=<n>]

Abbreviate the object name. When using *--hash*, you do not have to say *--hash --abbrev*; *--hash=n* would do.

-q , --quiet

Do not print any results to stdout. Can be used with *--verify* to silently check if a reference exists.

--exclude-existing[=<pattern>]

Make *git show-ref* act as a filter that reads refs from stdin of the form *^(?:<anything>\s)?<refname>(?:\^{})?$* and performs the following actions on each: (1) strip *^{}* at the end of line if any; (2) ignore if pattern is provided and does not head-match refname; (3) warn if refname is not a well-formed refname and skip; (4) ignore if refname is a ref that exists in the local repository; (5) otherwise output the line.

<pattern>…

Show references matching one or more patterns. Patterns are matched from the end of the full name, and only complete parts are matched, e.g. *master* matches *refs/heads/master*, *refs/remotes/origin/master*, *refs/tags/jedi/master* but not *refs/heads/mymaster* or *refs/remotes/master/jedi*.

OUTPUT

The output is in the format:

```
<oid> SP <ref> LF
```

For example,

```
$ git show-ref --head --dereference
832e76a9899f560a90ffd62ae2ce83bbeff58f54 HEAD
832e76a9899f560a90ffd62ae2ce83bbeff58f54 refs/heads/master
832e76a9899f560a90ffd62ae2ce83bbeff58f54 refs/heads/origin
3521017556c5de4159da4615a39fa4d5d2c279b5 refs/tags/v0.99.9c
6ddc0964034342519a87fe013781abf31c6db6ad refs/tags/v0.99.9c^{}
055e4ae3ae6eb344cbabf2a5256a49ea66040131 refs/tags/v1.0rc4
423325a2d24638ddcc82ce47be5e40be550f4507 refs/tags/v1.0rc4^{}
...
```

When using *--hash* (and not *--dereference*), the output is in the format:

```
<oid> LF
```

For example,

```
$ git show-ref --heads --hash
2e3ba0114a1f52b47df29743d6915d056be13278
185008ae97960c8d551adcd9e23565194651b5d1
03adf42c988195b50e1a1935ba5fcbc39b2b029b
...
```

EXAMPLES

To show all references called "master", whether tags or heads or anything else, and regardless of how deep in the reference naming hierarchy they are, use:

```
git show-ref master
```

This will show "refs/heads/master" but also "refs/remote/other-repo/master", if such references exist.

When using the --*verify* flag, the command requires an exact path:

```
git show-ref --verify refs/heads/master
```

will only match the exact branch called "master".

If nothing matches, *git show-ref* will return an error code of 1, and in the case of verification, it will show an error message.

For scripting, you can ask it to be quiet with the --*quiet* flag, which allows you to do things like

```
git show-ref --quiet --verify -- "refs/heads/$headname" ||
        echo "$headname is not a valid branch"
```

to check whether a particular branch exists or not (notice how we don't actually want to show any results, and we want to use the full refname for it in order to not trigger the problem with ambiguous partial matches).

To show only tags, or only proper branch heads, use --*tags* and/or --*heads* respectively (using both means that it shows tags and heads, but not other random references under the refs/ subdirectory).

To do automatic tag object dereferencing, use the -*d* or --*dereference* flag, so you can do

```
git show-ref --tags --dereference
```

to get a listing of all tags together with what they dereference.

FILES

*.git/refs/**, *.git/packed-refs*

SEE ALSO

Section G.3.52, “git-for-each-ref(1)”, Section G.3.76, “git-ls-remote(1)”, Section G.3.148, “git-update-ref(1)”, Section G.4.13, “gitrepository-layout(5)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.134. git-show(1)

2024-07-02
Revision History
2024-07-02

NAME

git-show - Show various types of objects

SYNOPSIS

git show [<options>] [<object>…]

DESCRIPTION

Shows one or more objects (blobs, trees, tags and commits).

For commits it shows the log message and textual diff. It also presents the merge commit in a special format as produced by *git diff-tree --cc*.

For tags, it shows the tag message and the referenced objects.

For trees, it shows the names (equivalent to *git ls-tree* with --name-only).

For plain blobs, it shows the plain contents.

Some options that *git log* command understands can be used to control how the changes the commit introduces are shown.

This manual page describes only the most frequently used options.

OPTIONS

<object>…

The names of objects to show (defaults to *HEAD*). For a more complete list of ways to spell object names, see "SPECIFYING REVISIONS" section in Section G.4.14, “gitrevisions(7)”.

--pretty[=<format>] , --format=<format>

Pretty-print the contents of the commit logs in a given format, where *<format>* can be one of *oneline*, *short*, *medium*, *full*, *fuller*, *reference*, *email*, *raw*, *format:<string>* and *tformat:<string>*. When *<format>* is none of the above, and has *%placeholder* in it, it acts as if *--pretty=tformat:<format>* were given.

See the "PRETTY FORMATS" section for some additional details for each format. When *=<format>* part is omitted, it defaults to *medium*.

Note: you can specify the default pretty format in the repository configuration (see Section G.3.29, “git-config(1)”).

--abbrev-commit

Instead of showing the full 40-byte hexadecimal commit object name, show a prefix that names the object uniquely. "--abbrev=<n>" (which also modifies diff output, if it is displayed) option can be used to specify the minimum length of the prefix.

This should make "--pretty=oneline" a whole lot more readable for people using 80-column terminals.

--no-abbrev-commit

Show the full 40-byte hexadecimal commit object name. This negates *--abbrev-commit*, either explicit or implied by other options such as "--oneline". It also overrides the *log.abbrevCommit* variable.

--oneline

This is a shorthand for "--pretty=oneline --abbrev-commit" used together.

--encoding=<encoding>

Commit objects record the character encoding used for the log message in their encoding header; this option can be used to tell the command to re-code the commit log message in the encoding preferred by the user. For non plumbing commands this defaults to UTF-8. Note that if an object claims to be encoded in *X* and we are outputting in *X*, we will output the object verbatim; this means that invalid sequences in the original commit may be copied to the output. Likewise, if iconv(3) fails to convert the commit, we will quietly output the original object verbatim.

--expand-tabs=<n> , --expand-tabs , --no-expand-tabs

Perform a tab expansion (replace each tab with enough spaces to fill to the next display column that is a multiple of *<n>*) in the log message before showing it in the output. *--expand-tabs* is a short-hand for *--expand-tabs=8*, and *--no-expand-tabs* is a short-hand for *--expand-tabs=0*, which disables tab expansion.

By default, tabs are expanded in pretty formats that indent the log message by 4 spaces (i.e. *medium*, which is the default, *full*, and *fuller*).

--notes[=<ref>]

Show the notes (see Section G.3.94, "git-notes(1)") that annotate the commit, when showing the commit log message. This is the default for *git log*, *git show* and *git whatchanged* commands when there is no *--pretty*, *--format*, or *--oneline* option given on the command line.

By default, the notes shown are from the notes refs listed in the *core.notesRef* and *notes.displayRef* variables (or corresponding environment overrides). See Section G.3.29, "git-config(1)" for more details.

With an optional *<ref>* argument, use the ref to find the notes to display. The ref can specify the full refname when it begins with *refs/notes/*; when it begins with *notes/*, *refs/* and otherwise *refs/notes/* is prefixed to form the full name of the ref.

Multiple --notes options can be combined to control which notes are being displayed. Examples: "--notes=foo" will show only notes from "refs/notes/foo"; "--notes=foo --notes" will show both notes from "refs/notes/foo" and from the default notes ref(s).

--no-notes

Do not show notes. This negates the above *--notes* option, by resetting the list of notes refs from which notes are shown. Options are parsed in the order given on the command line, so e.g. "--notes --notes=foo --no-notes --notes=bar" will only show notes from "refs/notes/bar".

--show-notes-by-default

Show the default notes unless options for displaying specific notes are given.

--show-notes[=<ref>] , --[no-]standard-notes

These options are deprecated. Use the above --notes/--no-notes options instead.

--show-signature

Check the validity of a signed commit object by passing the signature to *gpg --verify* and show the output.

PRETTY FORMATS

If the commit is a merge, and if the pretty-format is not *oneline*, *email* or *raw*, an additional line is inserted before the *Author:* line. This line begins with "Merge: " and the hashes of ancestral commits are printed, separated by spaces. Note that the listed commits may not necessarily be the list of the **direct** parent commits if you have limited your view of history: for example, if you are only interested in changes related to a certain directory or file.

There are several built-in formats, and you can define additional formats by setting a pretty.<name> config option to either another format name, or a *format:* string, as described below (see Section G.3.29, "git-config(1)"). Here are the details of the built-in formats:

- *oneline*

  ```
  <hash> <title-line>
  ```

  This is designed to be as compact as possible.

- *short*

```
commit <hash>
Author: <author>

<title-line>
```

- *medium*

```
commit <hash>
Author: <author>
Date:   <author-date>

<title-line>

<full-commit-message>
```

- *full*

```
commit <hash>
Author: <author>
Commit: <committer>

<title-line>

<full-commit-message>
```

- *fuller*

```
commit <hash>
Author:     <author>
AuthorDate: <author-date>
Commit:     <committer>
CommitDate: <committer-date>

<title-line>

<full-commit-message>
```

- *reference*

```
<abbrev-hash> (<title-line>, <short-author-date>)
```

This format is used to refer to another commit in a commit message and is the same as *--pretty='format:%C(auto)%h (%s, %ad)'*. By default, the date is formatted with *--date=short* unless another *--date* option is explicitly specified. As with any *format:* with format placeholders, its output is not affected by other options like *--decorate* and *--walk-reflogs*.

- *email*

```
From <hash> <date>
From: <author>
Date: <author-date>
Subject: [PATCH] <title-line>

<full-commit-message>
```

- *mboxrd*

Like *email*, but lines in the commit message starting with "From " (preceded by zero or more ">") are quoted with ">" so they aren't confused as starting a new commit.

- *raw*

The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the hashes are displayed in full, regardless of whether --abbrev or --no-abbrev are used, and *parents* information show the true parent commits, without taking grafts or history simplification into account. Note that this format affects the way commits are displayed, but not the way the diff is shown e.g. with *git log --raw*. To get full object names in a raw diff format, use *--no-abbrev*.

- *format:<format-string>*

  The *format:<format-string>* format allows you to specify which information you want to show. It works a little bit like printf format, with the notable exception that you get a newline with *%n* instead of *\n*.

  E.g, *format:"The author of %h was %an, %ar%nThe title was >>%s<<%n"* would show something like this:

```
The author of fe6e0ee was Junio C Hamano, 23 hours ago
The title was >>t4119: test autocomputing -p<n> for traditional diff
 input.<<
```

  The placeholders are:

  - Placeholders that expand to a single literal character:

    %n

    newline

    %%

    a raw *%*

    %x00

    %x followed by two hexadecimal digits is replaced with a byte with the hexadecimal digits' value (we will call this "literal formatting code" in the rest of this document).

  - Placeholders that affect formatting of later placeholders:

    %Cred

    switch color to red

    %Cgreen

    switch color to green

    %Cblue

    switch color to blue

    %Creset

    reset color

    %C(…)

    color specification, as described under Values in the "CONFIGURATION FILE" section of Section G.3.29, “git-config(1)”. By default, colors are shown only when enabled for log output (by *color.diff*, *color.ui*, or *--color*, and respecting the *auto* settings of the former if we are going to a terminal). *%C(auto,...)* is accepted as a historical synonym for the default (e.g., *%C(auto,red)*). Specifying *%C(always,...)* will show the colors even when color is not otherwise enabled (though consider just using *--color=always* to enable color for the whole output, including this format and anything else git might color). *auto* alone (i.e. *%C(auto)*) will turn on auto coloring on the next placeholders until the color is switched again.

%m

left (<), right (>) or boundary (-) mark

%w([<w>[,<i1>[,<i2>]]])

switch line wrapping, like the -w option of Section G.3.130, “git-shortlog(1)”.

%<(<N> [,trunc|ltrunc|mtrunc])

make the next placeholder take at least N column widths, padding spaces on the right if necessary. Optionally truncate (with ellipsis ..) at the left (ltrunc) *..ft*, the middle (mtrunc) *mi..le*, or the end (trunc) *rig..*, if the output is longer than N columns. Note 1: that truncating only works correctly with N >= 2. Note 2: spaces around the N and M (see below) values are optional. Note 3: Emojis and other wide characters will take two display columns, which may over-run column boundaries. Note 4: decomposed character combining marks may be misplaced at padding boundaries.

%<|(<M>)

make the next placeholder take at least until Mth display column, padding spaces on the right if necessary. Use negative M values for column positions measured from the right hand edge of the terminal window.

%>(<N>), %>|(<M>)

similar to *%<(<N>), %<|(<M>)* respectively, but padding spaces on the left

%>>(<N>), %>>|(<M>)

similar to *%>(<N>), %>|(<M>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces

%><(<N>), %><|(<M>)

similar to *%<(<N>), %<|(<M>)* respectively, but padding both sides (i.e. the text is centered)

- Placeholders that expand to information extracted from the commit:

%H

commit hash

%h

abbreviated commit hash

%T

tree hash

%t

abbreviated tree hash

%P

parent hashes

%p

abbreviated parent hashes

%an

author name

%aN

author name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ae

author email

%aE

author email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%al

author email local-part (the part before the @ sign)

%aL

author local-part (see *%al*) respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ad

author date (format respects --date= option)

%aD

author date, RFC2822 style

%ar

author date, relative

%at

author date, UNIX timestamp

%ai

author date, ISO 8601-like format

%aI

author date, strict ISO 8601 format

%as

author date, short format (*YYYY-MM-DD*)

%ah

author date, human style (like the *--date=human* option of Section G.3.120, “git-rev-list(1)”)

%cn

committer name

%cN

committer name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ce

committer email

%cE

committer email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%cl

committer email local-part (the part before the @ sign)

%cL

committer local-part (see *%cl*) respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%cd

committer date (format respects --date= option)

%cD

committer date, RFC2822 style

%cr

committer date, relative

%ct

committer date, UNIX timestamp

%ci

committer date, ISO 8601-like format

%cI

committer date, strict ISO 8601 format

%cs

committer date, short format (*YYYY-MM-DD*)

%ch

committer date, human style (like the *--date=human* option of Section G.3.120, “git-rev-list(1)”)

%d

ref names, like the --decorate option of Section G.3.74, “git-log(1)”

%D

ref names without the " (", ")" wrapping.

%(decorate[:<options>])

ref names with custom decorations. The *decorate* string may be followed by a colon and zero or more comma-separated options. Option values may contain literal formatting codes. These must be used for commas (*%x2C*) and closing parentheses (*%x29*), due to their role in the option syntax.

- *prefix=<value>*: Shown before the list of ref names. Defaults to " *(*".
- *suffix=<value>*: Shown after the list of ref names. Defaults to "*)*".
- *separator=<value>*: Shown between ref names. Defaults to ", ".
- *pointer=<value>*: Shown between HEAD and the branch it points to, if any. Defaults to " -> ".
- *tag=<value>*: Shown before tag names. Defaults to "*tag:* ".

For example, to produce decorations with no wrapping or tag annotations, and spaces as separators:

%(decorate:prefix=,suffix=,tag=,separator=)

%(describe[:<options>])

human-readable name, like Section G.3.39, “git-describe(1)”; empty string for undescribable commits. The *describe* string may be followed by a colon and zero or more comma-separated options. Descriptions can be inconsistent when tags are added or removed at the same time.

- *tags[=<bool-value>]*: Instead of only considering annotated tags, consider lightweight tags as well.
- *abbrev=<number>*: Instead of using the default number of hexadecimal digits (which will vary according to the number of objects in the repository with a default of 7) of the abbreviated object name, use <number> digits, or as many digits as needed to form a unique object name.
- *match=<pattern>*: Only consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix.
- *exclude=<pattern>*: Do not consider tags matching the given *glob(7)* pattern, excluding the "refs/tags/" prefix.

%S

ref name given on the command line by which the commit was reached (like *git log --source*), only works with *git log*

%e

encoding

%s

subject

%f

sanitized subject line, suitable for a filename

%b

body

%B

raw body (unwrapped subject and body)

%N

commit notes

%GG

raw verification message from GPG for a signed commit

%G?

show "G" for a good (valid) signature, "B" for a bad signature, "U" for a good signature with unknown validity, "X" for a good signature that has expired, "Y" for a good signature made by an expired key, "R" for a good signature made by a revoked key, "E" if the signature cannot be checked (e.g. missing key) and "N" for no signature

%GS

show the name of the signer for a signed commit

%GK

show the key used to sign a signed commit

%GF

show the fingerprint of the key used to sign a signed commit

%GP

show the fingerprint of the primary key whose subkey was used to sign a signed commit

%GT

show the trust level for the key used to sign a signed commit

%gD

reflog selector, e.g., *refs/stash@{1}* or *refs/stash@{2 minutes ago}*; the format follows the rules described for the *-g* option. The portion before the @ is the refname as given on the command line (so *git log -g refs/heads/master* would yield *refs/heads/master@{0}*).

%gd

shortened reflog selector; same as *%gD*, but the refname portion is shortened for human readability (so *refs/heads/master* becomes just *master*).

%gn

reflog identity name

%gN

reflog identity name (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%ge

reflog identity email

%gE

reflog identity email (respecting .mailmap, see Section G.3.130, “git-shortlog(1)” or Section G.3.9, “git-blame(1)”)

%gs

reflog subject

%(trailers[:<options>])

display the trailers of the body as interpreted by Section G.3.73, “git-interpret-trailers(1)”. The *trailers* string may be followed by a colon and zero or more comma-separated options. If any option is provided multiple times, the last occurrence wins.

- *key=<key>*: only show trailers with specified <key>. Matching is done case-insensitively and trailing colon is optional. If option is given multiple times trailer lines matching any of the keys are shown. This option automatically enables the *only* option so that non-trailer lines in the trailer block are hidden. If that is not desired it can be disabled with *only=false*. E.g., *%(trailers:key=Reviewed-by)* shows trailer lines with key *Reviewed-by*.
- *only[=<bool>]*: select whether non-trailer lines from the trailer block should be included.
- *separator=<sep>*: specify the separator inserted between trailer lines. Defaults to a line feed character. The string <sep> may contain the literal formatting codes described above. To use comma as separator one must use *%x2C* as it would otherwise be parsed as next option. E.g., *%(trailers:key=Ticket,separator=%x2C)* shows all trailer lines whose key is "Ticket" separated by a comma and a space.
- *unfold[=<bool>]*: make it behave as if interpret-trailer's *--unfold* option was given. E.g., *%(trailers:only,unfold=true)* unfolds and shows all trailer lines.
- *keyonly[=<bool>]*: only show the key part of the trailer.
- *valueonly[=<bool>]*: only show the value part of the trailer.
- *key_value_separator=<sep>*: specify the separator inserted between the key and value of each trailer. Defaults to ": ". Otherwise it shares the same semantics as *separator=<sep>* above.

> **Note**
>
> Some placeholders may depend on other options given to the revision traversal engine. For example, the *%g** reflog options will insert an empty string unless we are traversing reflog entries (e.g., by *git log -g*). The *%d* and *%D* placeholders will use the "short" decoration format if *--decorate* was not already provided on the command line.

The boolean options accept an optional value *[=<bool-value>]*. The values *true*, *false*, *on*, *off* etc. are all accepted. See the "boolean" sub-section in "EXAMPLES" in Section G.3.29, “git-config(1)”. If a boolean option is given with no value, it's enabled.

If you add a + (plus sign) after % of a placeholder, a line-feed is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

If you add a - (minus sign) after % of a placeholder, all consecutive line-feeds immediately preceding the expansion are deleted if and only if the placeholder expands to an empty string.

If you add a ` ` (space) after % of a placeholder, a space is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

- *tformat:*

  The *tformat:* format works exactly like *format:*, except that it provides "terminator" semantics instead of "separator" semantics. In other words, each commit has the message terminator character (usually a newline) appended, rather than a separator placed between entries. This means that the final entry of a single-line format will be properly terminated with a new line, just as the "oneline" format does. For example:

  ```
  $ git log -2 --pretty=format:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973 -- NO NEWLINE

  $ git log -2 --pretty=tformat:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973
  ```

  In addition, any unrecognized string that has a *%* in it is interpreted as if it has *tformat:* in front of it. For example, these two are equivalent:

  ```
  $ git log -2 --pretty=tformat:%h 4da45bef
  $ git log -2 --pretty=%h 4da45bef
  ```

DIFF FORMATTING

The options below can be used to change the way *git show* generates diff output.

-p , -u , --patch

Generate patch (see the section called “Generating patch text with -p”).

-s , --no-patch

Suppress all output from the diff machinery. Useful for commands like *git show* that show the patch by default to squelch their output, or to cancel the effect of options like *--patch*, *--stat* earlier on the command line in an alias.

-m

Show diffs for merge commits in the default format. This is similar to *--diff-merges=on*, except *-m* will produce no output unless *-p* is given as well.

-c

Produce combined diff output for merge commits. Shortcut for *--diff-merges=combined -p*.

--cc

Produce dense combined diff output for merge commits. Shortcut for *--diff-merges=dense-combined -p*.

--dd

Produce diff with respect to first parent for both merge and regular commits. Shortcut for *--diff-merges=first-parent -p*.

--remerge-diff

Produce remerge-diff output for merge commits. Shortcut for *--diff-merges=remerge -p*.

--no-diff-merges

Synonym for *--diff-merges=off*.

--diff-merges=<format>

Specify diff format to be used for merge commits. Default is *dense-combined* unless *--first-parent* is in use, in which case *first-parent* is the default.

The following formats are supported:

off, none

Disable output of diffs for merge commits. Useful to override implied value.

on, m

Make diff output for merge commits to be shown in the default format. The default format can be changed using *log.diffMerges* configuration variable, whose default value is *separate*.

first-parent, 1

Show full diff with respect to first parent. This is the same format as *--patch* produces for non-merge commits.

separate

Show full diff with respect to each of parents. Separate log entry and diff is generated for each parent.

combined, c

Show differences from each of the parents to the merge result simultaneously instead of showing pairwise diff between a parent and the result one at a time. Furthermore, it lists only files which were modified from all parents.

dense-combined, cc

Further compress output produced by *--diff-merges=combined* by omitting uninteresting hunks whose contents in the parents have only two variants and the merge result picks one of them without modification.

remerge, r

Remerge two-parent merge commits to create a temporary tree object--potentially containing files with conflict markers and such. A diff is then shown between that temporary tree and the actual merge commit.

The output emitted when this option is used is subject to change, and so is its interaction with other options (unless explicitly documented).

--combined-all-paths

This flag causes combined diffs (used for merge commits) to list the name of the file from all parents. It thus only has effect when *--diff-merges=[dense-]combined* is in use, and is likely only useful if filename changes are detected (i.e. when either rename or copy detection have been requested).

-U<n> , --unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies *--patch*.

--output=<file>

Output to a specific file instead of stdout.

--output-indicator-new=<char> , --output-indicator-old=<char> , --output-indicator-context=<char>

Specify the character used to indicate new, old or context lines in the generated patch. Normally they are +, - and ' ' respectively.

--raw

For each commit, show a summary of changes using the raw diff format. See the "RAW OUTPUT FORMAT" section of Section G.3.44, “git-diff(1)”. This is different from showing the log itself in raw format, which you can achieve with *--format=raw*.

--patch-with-raw

Synonym for *-p --raw*.

-t

Show the tree objects in the diff output.

--indent-heuristic

Enable the heuristic that shifts diff hunk boundaries to make patches easier to read. This is the default.

--no-indent-heuristic

Disable the indent heuristic.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--anchored=<text>

Generate a diff using the "anchored diff" algorithm.

This option may be specified more than once.

If a line exists in both the source and destination, exists only once, and starts with this text, this algorithm attempts to prevent it from appearing as a deletion or addition in the output. It uses the "patience diff" algorithm internally.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

default, myers

The basic greedy diff algorithm. Currently, this is the default.

minimal

Spend extra time to make sure the smallest possible diff is produced.

patience

Use "patience diff" algorithm when generating patches.

histogram

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured the *diff.algorithm* variable to a non-default value and want to use the default one, then you have to use *--diff-algorithm=default* option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by *<width>*. The width of the filename part can be limited by giving another width *<name-width>* after a comma or by setting *diff.statNameWidth=<width>*. The width of the graph part can be limited by using *--stat-graph-width=<width>* or by setting *diff.statGraphWidth=<width>*. Using *--stat* or *--stat-graph-width* affects all commands generating a stat graph, while setting *diff.statNameWidth* or *diff.statGraphWidth* does not affect *git format-patch*. By giving a third parameter *<count>*, you can limit the output to the first *<count>* lines, followed by ... if there are more.

These parameters can also be set individually with *--stat-width=<width>*, *--stat-name-width=<name-width>* and *--stat-count=<count>*.

--compact-summary

Output a condensed summary of extended header information such as file creations or deletions ("new" or "gone", optionally "+l" if it's a symlink) and mode changes ("+x" or "-x" for adding or removing executable bit respectively) in diffstat. The information is put between the filename part and the graph part. Implies *--stat*.

--numstat

Similar to *--stat*, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two - instead of saying *0 0*.

--shortstat

Output only the last line of the *--stat* format containing total number of modified files, as well as number of added and deleted lines.

-X[<param1,param2,…>] , --dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of *--dirstat* can be customized by passing it a comma separated list of parameters. The defaults are controlled by the *diff.dirstat* configuration variable (see Section G.3.29, “git-config(1)”). The following parameters are available:

changes

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

lines

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive *--dirstat* behavior than the *changes* behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other *--*stat* options.

files

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest *--dirstat* behavior, since it does not have to look at the file contents at all.

cumulative

Count changes in a child directory for the parent directory as well. Note that when using *cumulative*, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the *noncumulative* parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: *--dirstat=files,10,cumulative*.

--cumulative

Synonym for --dirstat=cumulative

--dirstat-by-file[=<param1,param2>…]

Synonym for --dirstat=files,<param1>,<param2>…

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for *-p --stat*.

-z

Separate the commits with NULs instead of newlines.

Also, when *--raw* or *--numstat* has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

--name-only

Show only the name of each changed file in the post-image tree. The file names are often encoded in UTF-8. For more information see the discussion about encoding in the Section G.3.74, “git-log(1)” manual page.

--name-status

Show only the name(s) and status of each changed file. See the description of the *--diff-filter* option on what the status letters mean. Just like *--name-only* the file names are often encoded in UTF-8.

--submodule[=<format>]

Specify how differences in submodules are shown. When specifying *--submodule=short* the *short* format is used. This format just shows the names of the commits at the beginning and end of the range. When *--submodule* or *--submodule=log* is specified, the *log* format is used. This format lists the commits in the range like Section G.3.141, “git-submodule(1)” *summary* does. When *--submodule=diff* is specified, the *diff* format is used. This format shows an inline diff of the changes in the submodule contents between the commit range. Defaults to *diff.submodule* or the *short* format if the config option is unset.

--color[=<when>]

Show colored diff. *--color* (i.e. without =*<when>*) is the same as *--color=always*. *<when>* can be one of *always*, *never*, or *auto*.

--no-color

Turn off colored diff. It is the same as *--color=never*.

--color-moved[=<mode>]

Moved lines of code are colored differently. The <mode> defaults to *no* if the option is not given and to *zebra* if the option with no mode is given. The mode must be one of:

no

Moved lines are not highlighted.

default

Is a synonym for *zebra*. This may change to a more sensible mode in the future.

plain

Any line that is added in one location and was removed in another location will be colored with *color.diff.newMoved*. Similarly *color.diff.oldMoved* will be used for removed lines that are added somewhere else in the diff. This mode picks up any moved line, but it is not very useful in a review to determine if a block of code was moved without permutation.

blocks

Blocks of moved text of at least 20 alphanumeric characters are detected greedily. The detected blocks are painted using either the *color.diff.{old,new}Moved* color. Adjacent blocks cannot be told apart.

zebra

Blocks of moved text are detected as in *blocks* mode. The blocks are painted using either the *color.diff.{old,new}Moved* color or *color.diff.{old,new}MovedAlternative*. The change between the two colors indicates that a new block was detected.

dimmed-zebra

Similar to *zebra*, but additional dimming of uninteresting parts of moved code is performed. The bordering lines of two adjacent blocks are considered interesting, the rest is uninteresting. *dimmed_zebra* is a deprecated synonym.

--no-color-moved

Turn off move detection. This can be used to override configuration settings. It is the same as *--color-moved=no*.

--color-moved-ws=<modes>

This configures how whitespace is ignored when performing the move detection for *--color-moved*. These modes can be given as a comma separated list:

no

Do not ignore whitespace when performing move detection.

ignore-space-at-eol

Ignore changes in whitespace at EOL.

ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

allow-indentation-change

Initially ignore any whitespace in the move detection, then group the moved code blocks only into a block if the change in whitespace is the same per line. This is incompatible with the other modes.

--no-color-moved-ws

Do not ignore whitespace when performing move detection. This can be used to override configuration settings. It is the same as *--color-moved-ws=no*.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see *--word-diff-regex* below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies *--color*.

plain

Show words as *[-removed-]* and *{+added+}*. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a +/-/` ` character at the beginning of the line and extending to the end of the line. Newlines in the input are represented by a tilde ~ on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies *--word-diff* unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append *|[^[:space:]]* to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, *--word-diff-regex=.* will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see Section G.4.2, “gitattributes(5)” or Section G.3.29, “git-config(1)”. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to *--word-diff=color* plus (if a regex was specified) *--word-diff-regex=<regex>*.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--[no-]rename-empty

Whether to use empty blobs as rename source.

--check

Warn if changes introduce conflict markers or whitespace errors. What are considered whitespace errors is controlled by *core.whitespace* configuration. By default, trailing whitespaces (including lines that consist solely of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors in the *context*, *old* or *new* lines of the diff. Multiple values are separated by comma, *none* resets previous values, *default* reset the list to *new* and *all* is a shorthand for *old,new,context*. When this option is not given, and the configuration variable *diff.wsErrorHighlight* is not set, only whitespace errors in *new* lines are highlighted. The whitespace errors are colored with *color.diff.whitespace*.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to *--full-index*, output a binary diff that can be applied with *git-apply*. Implies *--patch*.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show the shortest prefix that is at least *<n>* hexdigits long that uniquely refers the object. In diff-patch output format, *--full-index* takes higher precedence, i.e. if *--full-index* is specified, full blob names will be shown regardless of *--abbrev*. Non default number of digits can be specified with *--abbrev=<n>*.

-B[<n>][/<m>] , --break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything new, and the number *m* controls this aspect of the -B option (defaults to 60%). *-B/70%* specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number *n* controls this aspect of the -B option (defaults to 50%). *-B20%* specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>] , --find-renames[=<n>]

If generating diffs, detect and report renames for each commit. For following files across renames while traversing history, see *--follow*. If *n* is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, *-M90%* means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a *%* sign, the number is to be read as a

fraction, with a decimal point before it. I.e., *-M5* becomes 0.5, and is thus the same as *-M50%*. Similarly, *-M05* is the same as *-M5%*. To limit detection to exact renames, use *-M100%*. The default similarity index is 50%.

-C[<n>] , --find-copies[=<n>]

Detect copies as well as renames. See also *--find-copies-harder*. If *n* is specified, it has the same meaning as for *-M<n>*.

--find-copies-harder

For performance reasons, by default, *-C* option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one *-C* option has the same effect.

-D , --irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and */dev/null*. The resulting patch is not meant to be applied with *patch* or *git apply*; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lacks enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with *-B*, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The *-M* and *-C* options involve some preliminary steps that can detect subsets of renames/copies cheaply, followed by an exhaustive fallback portion that compares all remaining unpaired destinations to all relevant sources. (For renames, only remaining unpaired sources are relevant; for copies, all original sources are relevant.) For N sources and destinations, this exhaustive check is O(N^2). This option prevents the exhaustive portion of rename/copy detection from running if the number of source/destination files involved exceeds the specified number. Defaults to diff.renameLimit. Note that a value of 0 is treated as unlimited.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (*A*), Copied (*C*), Deleted (*D*), Modified (*M*), Renamed (*R*), have their type (i.e. regular file, symlink, submodule, …) changed (*T*), are Unmerged (*U*), are Unknown (*X*), or have had their pairing Broken (*B*). Any combination of the filter characters (including none) can be used. When * (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

Also, these upper-case letters can be downcased to exclude. E.g. *--diff-filter=ad* excludes added and deleted paths.

Note that not all diffs can feature all types. For instance, copied and renamed entries cannot appear if detection for those types is disabled.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter's use.

It is useful when you're looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into *-S*, and keep going until you get the very first version of the block.

Binary files are searched as well.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between *-S<regex> --pickaxe-regex* and *-G<regex>*, consider a commit with the following diff in the same file:

```
+    return frotz(nitfol, two->ptr, 1, 0);
...
-    hit = frotz(nitfol, mf2.ptr, 1, 0);
```

While *git log -G"frotz\(nitfol"* will show this commit, *git log -S"frotz\(nitfol" --pickaxe-regex* will not (because the number of occurrences of that string did not change).

Unless *--text* is supplied patches of binary files without a textconv filter will be ignored.

See the *pickaxe* entry in Section G.4.4, “gitdiffcore(7)” for more information.

--find-object=<object-id>

Look for differences that change the number of occurrences of the specified object. Similar to *-S*, just the argument is different in that it doesn't search for a specific string but for a specific object id.

The object can be a blob or a submodule commit. It implies the *-t* option in *git-log* to also find trees.

--pickaxe-all

When *-S* or *-G* finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to *-S* as an extended POSIX regular expression to match.

-O<orderfile>

Control the order in which files appear in the output. This overrides the *diff.orderFile* configuration variable (see Section G.3.29, “git-config(1)”). To cancel *diff.orderFile*, use *-O/dev/null*.

The output order is determined by the order of glob patterns in <orderfile>. All files with pathnames that match the first pattern are output first, all files with pathnames that match the second pattern (but not the first) are output next, and so on. All files with pathnames that do not match any pattern are output last, as if there was an implicit match-all pattern at the end of the file. If multiple pathnames have the same rank (they match the same pattern but no earlier patterns), their output order relative to each other is the normal order.

<orderfile> is parsed as follows:

- Blank lines are ignored, so they can be used as separators for readability.
- Lines starting with a hash ("#") are ignored, so they can be used for comments. Add a backslash ("\") to the beginning of the pattern if it starts with a hash.
- Each other line contains a single pattern.

Patterns have the same syntax and semantics as patterns used for fnmatch(3) without the FNM_PATHNAME flag, except a pathname also matches a pattern if removing any number of the final pathname components matches the pattern. For example, the pattern "*foo*bar*" matches "*fooasdfbar*" and "*foo/bar/baz/asdf*" but not "*foobarx*".

--skip-to=<file> , --rotate-to=<file>

Discard the files before the named <file> from the output (i.e. *skip to*), or move them to the end of the output (i.e. *rotate to*). These options were invented primarily for the use of the *git difftool* command, and may not be very useful otherwise.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>] , --no-relative

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument. *--no-relative* can be used to countermand both *diff.relative* config option and previous *--relative*.

-a , --text

Treat all files as text.

--ignore-cr-at-eol

Ignore carriage-return at the end of line when doing a comparison.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b , --ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w , --ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

-I<regex> , --ignore-matching-lines=<regex>

Ignore changes whose all lines match <regex>. This option may be specified more than once.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other. Defaults to *diff.interHunkContext* or 0 if the config option is unset.

-W , --function-context

Show whole function as context lines for each change. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with Section G.4.2, “gitattributes(5)”, you need to use this option with Section G.3.74, “git-log(1)” and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv , --no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See Section G.4.2, “gitattributes(5)” for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for Section G.3.44, “git-diff(1)” and Section G.3.74, “git-log(1)”, but not for Section G.3.54, “git-format-patch(1)” or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in Section G.3.29, “git-config(1)” or Section G.4.10, “gitmodules(5)”. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

--default-prefix

Use the default source and destination prefixes ("a/" and "b/"). This overrides configuration variables such as *diff.noprefix*, *diff.srcPrefix*, *diff.dstPrefix*, and *diff.mnemonicPrefix* (see *git-config*(1)).

--line-prefix=<prefix>

Prepend an additional prefix to every line of output.

--ita-invisible-in-index

By default entries added by "git add -N" appear as an existing empty file in "git diff" and a new file in "git diff --cached". This option makes the entry appear as a new file in "git diff" and non-existent in "git diff --cached". This option could be reverted with *--ita-visible-in-index*. Both options are experimental and could be removed in future.

For more detailed explanation on these common options, see also Section G.4.4, “gitdiffcore(7)”.

Generating patch text with -p

Running Section G.3.44, “git-diff(1)”, Section G.3.74, “git-log(1)”, Section G.3.134, “git-show(1)”, Section G.3.42, “git-diff-index(1)”, Section G.3.43, “git-diff-tree(1)”, or Section G.3.41, “git-diff-files(1)” with the *-p* option produces patch text. You can customize the creation of patch text via the *GIT_EXTERNAL_DIFF* and the *GIT_DIFF_OPTS* environment variables (see Section G.3.1, “git(1)”), and the *diff* attribute (see Section G.4.2, “gitattributes(5)”).

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded by a "git diff" header that looks like this:

   ```
   diff --git a/file1 b/file2
   ```

   The *a/* and *b/* filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, */dev/null* is *not* used in place of the *a/* or *b/* filenames.

   When a rename/copy is involved, *file1* and *file2* show the name of the source file of the rename/copy and the name of the file that the rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

```
old mode <mode>
new mode <mode>
deleted file mode <mode>
new file mode <mode>
copy from <path>
copy to <path>
rename from <path>
rename to <path>
similarity index <number>
dissimilarity index <number>
index <hash>..<hash> <mode>
```

File modes are printed as 6-digit octal numbers including the file type and file permission bits.

Path names in extended headers do not include the *a/* and *b/* prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the blob object names before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. Pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

4. All the *file1* files in the output refer to files before the commit, and all the *file2* files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

5. Hunk headers mention the name of the function to which the hunk applies. See "Defining a custom hunk-header" in Section G.4.2, “gitattributes(5)” for details of how to tailor this to specific languages.

Combined diff format

Any diff-generating command can take the *-c* or *--cc* option to produce a *combined diff* when showing a merge. This is the default format when showing merges with Section G.3.44, “git-diff(1)” or Section G.3.134, “git-show(1)”. Note also that you can give suitable *--diff-merges* option to any of these commands to force generation of diffs in a specific format.

A "combined diff" format looks like this:

```
diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
        return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
```

```
++static void describe(char *arg, int last_one)
  {
 +      unsigned char sha1[20];
 +      struct commit *cmit;
        struct commit_list *list;
        static int initialized = 0;
        struct commit_name *n;

 +      if (get_sha1(arg, sha1) < 0)
 +              usage(describe_usage);
 +      cmit = lookup_commit_reference(sha1);
 +      if (!cmit)
 +              usage(describe_usage);
 +
        if (!initialized) {
                initialized = 1;
                for_each_ref(get_name);
```

1. It is preceded by a "git diff" header, that looks like this (when the *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when the *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index <hash>,<hash>..<hash>
   mode <mode>,<mode>..<mode>
   new file mode <mode>
   deleted file mode <mode>,<mode>
   ```

   The *mode <mode>,<mode>..<mode>* line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected content movement (renames and copying detection) are designed to work with the diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by a two-line from-file/to-file header:

   ```
   --- a/file
   +++ b/file
   ```

   Similar to the two-line header for the traditional *unified* diff format, */dev/null* is used to signal created or deleted files.

   However, if the --combined-all-paths option is provided, instead of a two-line from-file/to-file, you get an N +1 line from-file/to-file header, where N is the number of parents in the merge commit:

   ```
   --- a/file
   --- a/file
   --- a/file
   +++ b/file
   ```

   This extended format can be useful if rename or copy detection is active, to allow you to see the original name of the file in different parents.

4. Chunk header format is modified to prevent people from accidentally feeding it to *patch -p1*. Combined diff format was created for review of merge commit changes, and was not meant to be applied. The change is similar to the change in the extended *index* header:

   ```
   @@@ <from-file-range> <from-file-range> <to-file-range> @@@
   ```

There are (number of parents + 1) @ characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has - (minus -- appears in A but removed in B), + (plus -- missing in A but added to B), or " " (space -- unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X's line is different from it.

A - character in the column N means that the line appears in fileN but it does not appear in the result. A + character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two - removals from both file1 and file2, plus ++ to mean one line that was added does not appear in either file1 or file2). Also, eight other lines are the same from file1 but do not appear in file2 (hence prefixed with +).

When shown by *git diff-tree -c*, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by *git diff-files -c*, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

EXAMPLES

git show v1.0.0

Shows the tag *v1.0.0*, along with the object the tag points at.

git show v1.0.0^{tree}

Shows the tree pointed to by the tag *v1.0.0*.

git show -s --format=%s v1.0.0^{commit}

Shows the subject of the commit pointed to by the tag *v1.0.0*.

git show next~10:Documentation/README

Shows the contents of the file *Documentation/README* as they were current in the 10th last commit of the branch *next*.

git show master:Makefile master:t/Makefile

Concatenates the contents of said Makefiles in the head of the branch *master*.

DISCUSSION

Git is to some extent character encoding agnostic.

- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- Path names are encoded in UTF-8 normalization form C. This applies to tree objects, the index file, ref names, as well as path names in command line arguments, environment variables and config files (*.git/config* (see Section G.3.29, “git-config(1)”), Section G.4.5, “gitignore(5)”, Section G.4.2, “gitattributes(5)” and Section G.4.10, “gitmodules(5)”).

  Note that Git at the core level treats path names simply as sequences of non-NUL bytes, there are no path name encoding conversions (except on Mac and Windows). Therefore, using non-ASCII path names will mostly work even on platforms and file systems that use legacy extended ASCII encodings. However, repositories created on such systems will not work properly on UTF-8-based systems (e.g. Linux, Mac, Windows) and vice versa. Additionally, many Git-based tools simply assume path names to be UTF-8 and will fail to display other encodings correctly.

- Commit log messages are typically encoded in UTF-8, but other extended ASCII encodings are also supported. This includes ISO-8859-x, CP125x and many others, but *not* UTF-16/32, EBCDIC and CJK multi-byte encodings (GBK, Shift-JIS, Big5, EUC-x, CP9xx etc.).

Although we encourage that the commit log messages are encoded in UTF-8, both the core and Git Porcelain are designed not to force UTF-8 on projects. If all participants of a particular project find it more convenient to use legacy encodings, Git does not forbid it. However, there are a few things to keep in mind.

1. *git commit* and *git commit-tree* issue a warning if the commit log message given to it does not look like a valid UTF-8 string, unless you explicitly say your project uses a legacy encoding. The way to say this is to have *i18n.commitEncoding* in *.git/config* file, like this:

   ```
   [i18n]
           commitEncoding = ISO-8859-1
   ```

   Commit objects created with the above setting record the value of *i18n.commitEncoding* in their *encoding* header. This is to help other people who look at them later. Lack of this header implies that the commit log message is encoded in UTF-8.

2. *git log*, *git show*, *git blame* and friends look at the *encoding* header of a commit object, and try to re-code the log message into UTF-8 unless otherwise specified. You can specify the desired output encoding with *i18n.logOutputEncoding* in *.git/config* file, like this:

   ```
   [i18n]
           logOutputEncoding = ISO-8859-1
   ```

   If you do not have this configuration variable, the value of *i18n.commitEncoding* is used instead.

Note that we deliberately chose not to re-code the commit log message when a commit is made to force UTF-8 at the commit object level, because re-coding to UTF-8 is not necessarily a reversible operation.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.135. git-sparse-checkout(1)

2024-07-02
Revision History
2024-07-02

NAME

git-sparse-checkout - Reduce your working tree to a subset of tracked files

SYNOPSIS

git sparse-checkout (init | list | set | add | reapply | disable | check-rules) [<options>]

DESCRIPTION

This command is used to create sparse checkouts, which change the working tree from having all tracked files present to only having a subset of those files. It can also switch which subset of files are present, or undo and go back to having all tracked files present in the working copy.

The subset of files is chosen by providing a list of directories in cone mode (the default), or by providing a list of patterns in non-cone mode.

When in a sparse-checkout, other Git commands behave a bit differently. For example, switching branches will not update paths outside the sparse-checkout directories/patterns, and *git commit -a* will not record paths outside the sparse-checkout directories/patterns as deleted.

THIS COMMAND IS EXPERIMENTAL. ITS BEHAVIOR, AND THE BEHAVIOR OF OTHER COMMANDS IN THE PRESENCE OF SPARSE-CHECKOUTS, WILL LIKELY CHANGE IN THE FUTURE.

COMMANDS

list

Describe the directories or patterns in the sparse-checkout file.

set

Enable the necessary sparse-checkout config settings (*core.sparseCheckout*, *core.sparseCheckoutCone*, and *index.sparse*) if they are not already set to the desired values, populate the sparse-checkout file from the list of arguments following the *set* subcommand, and update the working directory to match.

To ensure that adjusting the sparse-checkout settings within a worktree does not alter the sparse-checkout settings in other worktrees, the *set* subcommand will upgrade your repository config to use worktree-specific config if not already present. The sparsity defined by the arguments to the *set* subcommand are stored in the worktree-specific sparse-checkout file. See Section G.3.159, “git-worktree(1)” and the documentation of *extensions.worktreeConfig* in Section G.3.29, “git-config(1)” for more details.

When the *--stdin* option is provided, the directories or patterns are read from standard in as a newline-delimited list instead of from the arguments.

By default, the input list is considered a list of directories, matching the output of *git ls-tree -d --name-only*. This includes interpreting pathnames that begin with a double quote (") as C-style quoted strings. Note that all files under the specified directories (at any depth) will be included in the sparse checkout, as well as files that are siblings of either the given directory or any of its ancestors (see *CONE PATTERN SET* below for more details). In the past, this was not the default, and *--cone* needed to be specified or *core.sparseCheckoutCone* needed to be enabled.

When *--no-cone* is passed, the input list is considered a list of patterns. This mode has a number of drawbacks, including not working with some options like *--sparse-index*. As explained in the "Non-cone Problems" section below, we do not recommend using it.

Use the *--[no-]sparse-index* option to use a sparse index (the default is to not use it). A sparse index reduces the size of the index to be more closely aligned with your sparse-checkout definition. This can have significant performance advantages for commands such as *git status* or *git add*. This feature is still experimental. Some commands might be slower with a sparse index until they are properly integrated with the feature.

WARNING: Using a sparse index requires modifying the index in a way that is not completely understood by external tools. If you have trouble with this compatibility, then run *git sparse-checkout init --no-sparse-index* to rewrite your index to not be sparse. Older versions of Git will not understand the sparse directory entries index extension and may fail to interact with your repository until it is disabled.

add

Update the sparse-checkout file to include additional directories (in cone mode) or patterns (in non-cone mode). By default, these directories or patterns are read from the command-line arguments, but they can be read from stdin using the *--stdin* option.

reapply

Reapply the sparsity pattern rules to paths in the working tree. Commands like merge or rebase can materialize paths to do their work (e.g. in order to show you a conflict), and other sparse-checkout commands might fail to sparsify an individual file (e.g. because it has unstaged changes or conflicts). In such cases, it can make sense to run *git sparse-checkout reapply* later after cleaning up affected paths (e.g. resolving conflicts, undoing or committing changes, etc.).

The *reapply* command can also take *--[no-]cone* and *--[no-]sparse-index* flags, with the same meaning as the flags from the *set* command, in order to change which sparsity mode you are using without needing to also respecify all sparsity paths.

disable

Disable the *core.sparseCheckout* config setting, and restore the working directory to include all files.

init

Deprecated command that behaves like *set* with no specified paths. May be removed in the future.

Historically, *set* did not handle all the necessary config settings, which meant that both *init* and *set* had to be called. Invoking both meant the *init* step would first remove nearly all tracked files (and in cone mode, ignored files too), then the *set* step would add many of the tracked files (but not ignored files) back. In addition to the lost files, the performance and UI of this combination was poor.

Also, historically, *init* would not actually initialize the sparse-checkout file if it already existed. This meant it was possible to return to a sparse-checkout without remembering which paths to pass to a subsequent *set* or *add* command. However, *--cone* and *--sparse-index* options would not be remembered across the disable command, so the easy restore of calling a plain *init* decreased in utility.

check-rules

Check whether sparsity rules match one or more paths.

By default *check-rules* reads a list of paths from stdin and outputs only the ones that match the current sparsity rules. The input is expected to consist of one path per line, matching the output of *git ls-tree --name-only* including that pathnames that begin with a double quote (") are interpreted as C-style quoted strings.

When called with the *--rules-file <file>* flag the input files are matched against the sparse checkout rules found in *<file>* instead of the current ones. The rules in the files are expected to be in the same form as accepted by *git sparse-checkout set --stdin* (in particular, they must be newline-delimited).

By default, the rules passed to the *--rules-file* option are interpreted as cone mode directories. To pass non-cone mode patterns with *--rules-file*, combine the option with the *--no-cone* option.

When called with the *-z* flag, the format of the paths input on stdin as well as the output paths are \0 terminated and not quoted. Note that this does not apply to the format of the rules passed with the *--rules-file* option.

EXAMPLES

git sparse-checkout set MY/DIR1 SUB/DIR2

Change to a sparse checkout with all files (at any depth) under MY/DIR1/ and SUB/DIR2/ present in the working copy (plus all files immediately under MY/ and SUB/ and the toplevel directory). If already in a sparse checkout, change which files are present in the working copy to this new selection. Note that this command will also delete all ignored files in any directory that no longer has either tracked or non-ignored-untracked files present.

git sparse-checkout disable

Repopulate the working directory with all files, disabling sparse checkouts.

git sparse-checkout add SOME/DIR/ECTORY

Add all files under SOME/DIR/ECTORY/ (at any depth) to the sparse checkout, as well as all files immediately under SOME/DIR/ and immediately under SOME/. Must already be in a sparse checkout before using this command.

git sparse-checkout reapply

It is possible for commands to update the working tree in a way that does not respect the selected sparsity directories. This can come from tools external to Git writing files, or even affect Git commands because of either special cases (such as hitting conflicts when merging/rebasing), or because some commands didn't fully

support sparse checkouts (e.g. the old *recursive* merge backend had only limited support). This command reapplies the existing sparse directory specifications to make the working directory match.

INTERNALS -- SPARSE CHECKOUT

"Sparse checkout" allows populating the working directory sparsely. It uses the skip-worktree bit (see Section G.3.147, “git-update-index(1)”) to tell Git whether a file in the working directory is worth looking at. If the skip-worktree bit is set, and the file is not present in the working tree, then its absence is ignored. Git will avoid populating the contents of those files, which makes a sparse checkout helpful when working in a repository with many files, but only a few are important to the current user.

The *$GIT_DIR/info/sparse-checkout* file is used to define the skip-worktree reference bitmap. When Git updates the working directory, it updates the skip-worktree bits in the index based on this file. The files matching the patterns in the file will appear in the working directory, and the rest will not.

INTERNALS -- NON-CONE PROBLEMS

The *$GIT_DIR/info/sparse-checkout* file populated by the *set* and *add* subcommands is defined to be a bunch of patterns (one per line) using the same syntax as *.gitignore* files. In cone mode, these patterns are restricted to matching directories (and users only ever need supply or see directory names), while in non-cone mode any gitignore-style pattern is permitted. Using the full gitignore-style patterns in non-cone mode has a number of shortcomings:

- Fundamentally, it makes various worktree-updating processes (pull, merge, rebase, switch, reset, checkout, etc.) require O(N*M) pattern matches, where N is the number of patterns and M is the number of paths in the index. This scales poorly.
- Avoiding the scaling issue has to be done via limiting the number of patterns via specifying leading directory name or glob.
- Passing globs on the command line is error-prone as users may forget to quote the glob, causing the shell to expand it into all matching files and pass them all individually along to sparse-checkout set/add. While this could also be a problem with e.g. "git grep -- *.c", mistakes with grep/log/status appear in the immediate output. With sparse-checkout, the mistake gets recorded at the time the sparse-checkout command is run and might not be problematic until the user later switches branches or rebases or merges, thus putting a delay between the user's error and when they have a chance to catch/notice it.
- Related to the previous item, sparse-checkout has an *add* subcommand but no *remove* subcommand. Even if a *remove* subcommand were added, undoing an accidental unquoted glob runs the risk of "removing too much", as it may remove entries that had been included before the accidental add.
- Non-cone mode uses gitignore-style patterns to select what to **include** (with the exception of negated patterns), while .gitignore files use gitignore-style patterns to select what to **exclude** (with the exception of negated patterns). The documentation on gitignore-style patterns usually does not talk in terms of matching or non-matching, but on what the user wants to "exclude". This can cause confusion for users trying to learn how to specify sparse-checkout patterns to get their desired behavior.
- Every other git subcommand that wants to provide "special path pattern matching" of some sort uses pathspecs, but non-cone mode for sparse-checkout uses gitignore patterns, which feels inconsistent.
- It has edge cases where the "right" behavior is unclear. Two examples:

```
First, two users are in a subdirectory, and the first runs
   git sparse-checkout set '/toplevel-dir/*.c'
while the second runs
   git sparse-checkout set relative-dir
Should those arguments be transliterated into
   current/subdirectory/toplevel-dir/*.c
and
   current/subdirectory/relative-dir
```

```
before inserting into the sparse-checkout file?  The user who typed
the first command is probably aware that arguments to set/add are
supposed to be patterns in non-cone mode, and probably would not be
happy with such a transliteration.  However, many gitignore-style
patterns are just paths, which might be what the user who typed the
second command was thinking, and they'd be upset if their argument
wasn't transliterated.

Second, what should bash-completion complete on for set/add commands
for non-cone users?  If it suggests paths, is it exacerbating the
problem above?  Also, if it suggests paths, what if the user has a
file or directory that begins with either a '!' or '#' or has a '*',
'\', '?', '[', or ']' in its name?  And if it suggests paths, will
it complete "/pro" to "/proc" (in the root filesystem) rather than to
"/progress.txt" in the current directory?  (Note that users are
likely to want to start paths with a leading '/' in non-cone mode,
for the same reason that .gitignore files often have one.)
Completing on files or directories might give nasty surprises in
all these cases.
```

- The excessive flexibility made other extensions essentially impractical. *--sparse-index* is likely impossible in non-cone mode; even if it is somehow feasible, it would have been far more work to implement and may have been too slow in practice. Some ideas for adding coupling between partial clones and sparse checkouts are only practical with a more restricted set of paths as well.

For all these reasons, non-cone mode is deprecated. Please switch to using cone mode.

INTERNALS -- CONE MODE HANDLING

The "cone mode", which is the default, lets you specify only what directories to include. For any directory specified, all paths below that directory will be included, and any paths immediately under leading directories (including the toplevel directory) will also be included. Thus, if you specified the directory Documentation/technical/ then your sparse checkout would contain:

- all files in the toplevel-directory
- all files immediately under Documentation/
- all files at any depth under Documentation/technical/

Also, in cone mode, even if no directories are specified, then the files in the toplevel directory will be included.

When changing the sparse-checkout patterns in cone mode, Git will inspect each tracked directory that is not within the sparse-checkout cone to see if it contains any untracked files. If all of those files are ignored due to the *.gitignore* patterns, then the directory will be deleted. If any of the untracked files within that directory is not ignored, then no deletions will occur within that directory and a warning message will appear. If these files are important, then reset your sparse-checkout definition so they are included, use *git add* and *git commit* to store them, then remove any remaining files manually to ensure Git can behave optimally.

See also the "Internals -- Cone Pattern Set" section to learn how the directories are transformed under the hood into a subset of the Full Pattern Set of sparse-checkout.

INTERNALS -- FULL PATTERN SET

The full pattern set allows for arbitrary pattern matches and complicated inclusion/exclusion rules. These can result in O(N*M) pattern matches when updating the index, where N is the number of patterns and M is the number of paths in the index. To combat this performance issue, a more restricted pattern set is allowed when *core.sparseCheckoutCone* is enabled.

The sparse-checkout file uses the same syntax as *.gitignore* files; see Section G.4.5, “gitignore(5)” for details. Here, though, the patterns are usually being used to select which files to include rather than which files to exclude.

(However, it can get a bit confusing since gitignore-style patterns have negations defined by patterns which begin with a *!*, so you can also select files to *not* include.)

For example, to select everything, and then to remove the file *unwanted* (so that every file will appear in your working tree except the file named *unwanted*):

```
git sparse-checkout set --no-cone '/*' '!unwanted'
```

These patterns are just placed into the *$GIT_DIR/info/sparse-checkout* as-is, so the contents of that file at this point would be

```
/*
!unwanted
```

See also the "Sparse Checkout" section of Section G.3.106, “git-read-tree(1)” to learn more about the gitignore-style patterns used in sparse checkouts.

INTERNALS -- CONE PATTERN SET

In cone mode, only directories are accepted, but they are translated into the same gitignore-style patterns used in the full pattern set. We refer to the particular patterns used in those mode as being of one of two types:

1. **Recursive:** All paths inside a directory are included.

2. **Parent:** All files immediately inside a directory are included.

Since cone mode always includes files at the toplevel, when running *git sparse-checkout set* with no directories specified, the toplevel directory is added as a parent pattern. At this point, the sparse-checkout file contains the following patterns:

```
/*
!/*/
```

This says "include everything immediately under the toplevel directory, but nothing at any level below that."

When in cone mode, the *git sparse-checkout set* subcommand takes a list of directories. The command *git sparse-checkout set A/B/C* sets the directory *A/B/C* as a recursive pattern, the directories *A* and *A/B* are added as parent patterns. The resulting sparse-checkout file is now

```
/*
!/*/
/A/
!/A/*/
/A/B/
!/A/B/*/
/A/B/C/
```

Here, order matters, so the negative patterns are overridden by the positive patterns that appear lower in the file.

Unless *core.sparseCheckoutCone* is explicitly set to *false*, Git will parse the sparse-checkout file expecting patterns of these types. Git will warn if the patterns do not match. If the patterns do match the expected format, then Git will use faster hash-based algorithms to compute inclusion in the sparse-checkout. If they do not match, git will behave as though *core.sparseCheckoutCone* was false, regardless of its setting.

In the cone mode case, despite the fact that full patterns are written to the $GIT_DIR/info/sparse-checkout file, the *git sparse-checkout list* subcommand will list the directories that define the recursive patterns. For the example sparse-checkout file above, the output is as follows:

```
$ git sparse-checkout list
A/B/C
```

If *core.ignoreCase=true*, then the pattern-matching algorithm will use a case-insensitive check. This corrects for case mismatched filenames in the *git sparse-checkout set* command to reflect the expected cone in the working directory.

INTERNALS -- SUBMODULES

If your repository contains one or more submodules, then submodules are populated based on interactions with the *git submodule* command. Specifically, *git submodule init* -- <*path*> will ensure the submodule at <*path*> is present, while *git submodule deinit [-f]* -- <*path*> will remove the files for the submodule at <*path*> (including any untracked files, uncommitted changes, and unpushed history). Similar to how sparse-checkout removes files from the working tree but still leaves entries in the index, deinitialized submodules are removed from the working directory but still have an entry in the index.

Since submodules may have unpushed changes or untracked files, removing them could result in data loss. Thus, changing sparse inclusion/exclusion rules will not cause an already checked out submodule to be removed from the working copy. Said another way, just as *checkout* will not cause submodules to be automatically removed or initialized even when switching between branches that remove or add submodules, using *sparse-checkout* to reduce or expand the scope of "interesting" files will not cause submodules to be automatically deinitialized or initialized either.

Further, the above facts mean that there are multiple reasons that "tracked" files might not be present in the working copy: sparsity pattern application from sparse-checkout, and submodule initialization state. Thus, commands like *git grep* that work on tracked files in the working copy may return results that are limited by either or both of these restrictions.

SEE ALSO

Section G.3.106, “git-read-tree(1)” Section G.4.5, “gitignore(5)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.136. git-stage(1)

2024-07-02
Revision History

2024-07-02

NAME

git-stage - Add file contents to the staging area

SYNOPSIS

git stage <arg>…

DESCRIPTION

This is a synonym for Section G.3.2, “git-add(1)”. Please refer to the documentation of that command.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.137. git-stash(1)

2024-07-02
Revision History

2024-07-02

NAME

git-stash - Stash the changes in a dirty working directory away

SYNOPSIS

git stash list [<log-options>]
git stash show [-u | --include-untracked | --only-untracked] [<diff-options>] [<stash>]
git stash drop [-q | --quiet] [<stash>]
git stash pop [--index] [-q | --quiet] [<stash>]
git stash apply [--index] [-q | --quiet] [<stash>]
git stash branch <branchname> [<stash>]
git stash [push [-p | --patch] [-S | --staged] [-k | --[no-]keep-index] [-q | --quiet]
[-u | --include-untracked] [-a | --all] [(-m | --message) <message>]
[--pathspec-from-file=<file> [--pathspec-file-nul]]
[--] [<pathspec>…]]
git stash save [-p | --patch] [-S | --staged] [-k | --[no-]keep-index] [-q | --quiet]
[-u | --include-untracked] [-a | --all] [<message>]
git stash clear
git stash create [<message>]
git stash store [(-m | --message) <message>] [-q | --quiet] <commit>

DESCRIPTION

Use *git stash* when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the *HEAD* commit.

The modifications stashed away by this command can be listed with *git stash list*, inspected with *git stash show*, and restored (potentially on top of a different commit) with *git stash apply*. Calling *git stash* without any arguments is equivalent to *git stash push*. A stash is by default listed as "WIP on *branchname* …", but you can give a more descriptive message on the command line when you create one.

The latest stash you created is stored in *refs/stash*; older stashes are found in the reflog of this reference and can be named using the usual reflog syntax (e.g. *stash@{0}* is the most recently created stash, *stash@{1}* is the one before it, *stash@{2.hours.ago}* is also possible). Stashes may also be referenced by specifying just the stash index (e.g. the integer *n* is equivalent to *stash@{n}*).

COMMANDS

push [-p|--patch] [-S|--staged] [-k|--[no-]keep-index] [-u|--include-untracked] [-a|--all] [-q|--quiet] [(-m|--message) <message>] [--pathspec-from-file=<file> [--pathspec-file-nul]] [--] [<pathspec>…]

Save your local modifications to a new *stash entry* and roll them back to HEAD (in the working tree and in the index). The <message> part is optional and gives the description along with the stashed state.

For quickly making a snapshot, you can omit "push". In this mode, non-option arguments are not allowed to prevent a misspelled subcommand from making an unwanted stash entry. The two exceptions to this are *stash -p* which acts as alias for *stash push -p* and pathspec elements, which are allowed after a double hyphen -- for disambiguation.

save [-p|--patch] [-S|--staged] [-k|--[no-]keep-index] [-u|--include-untracked] [-a|--all] [-q|--quiet] [<message>]

This option is deprecated in favour of *git stash push*. It differs from "stash push" in that it cannot take pathspec. Instead, all non-option arguments are concatenated to form the stash message.

list [<log-options>]

List the stash entries that you currently have. Each *stash entry* is listed with its name (e.g. *stash@{0}* is the latest entry, *stash@{1}* is the one before, etc.), the name of the branch that was current when the entry was made, and a short description of the commit the entry was based on.

```
stash@{0}: WIP on submit: 6ebd0e2... Update git-stash documentation
stash@{1}: On master: 9cc0589... Add git-stash
```

The command takes options applicable to the *git log* command to control what is shown and how. See Section G.3.74, “git-log(1)”.

show [-u|--include-untracked|--only-untracked] [<diff-options>] [<stash>]

Show the changes recorded in the stash entry as a diff between the stashed contents and the commit back when the stash entry was first created. By default, the command shows the diffstat, but it will accept any format known to *git diff* (e.g., *git stash show -p stash@{1}* to view the second most recent entry in patch form). If no *<diff-option>* is provided, the default behavior will be given by the *stash.showStat*, and *stash.showPatch* config variables. You can also use *stash.showIncludeUntracked* to set whether *--include-untracked* is enabled by default.

pop [--index] [-q|--quiet] [<stash>]

Remove a single stashed state from the stash list and apply it on top of the current working tree state, i.e., do the inverse operation of *git stash push*. The working directory must match the index.

Applying the state can fail with conflicts; in this case, it is not removed from the stash list. You need to resolve the conflicts by hand and call *git stash drop* manually afterwards.

apply [--index] [-q|--quiet] [<stash>]

Like *pop*, but do not remove the state from the stash list. Unlike *pop*, *<stash>* may be any commit that looks like a commit created by *stash push* or *stash create*.

branch <branchname> [<stash>]

Creates and checks out a new branch named *<branchname>* starting from the commit at which the *<stash>* was originally created, applies the changes recorded in *<stash>* to the new working tree and index. If that succeeds, and *<stash>* is a reference of the form *stash@{<revision>}*, it then drops the *<stash>*.

This is useful if the branch on which you ran *git stash push* has changed enough that *git stash apply* fails due to conflicts. Since the stash entry is applied on top of the commit that was HEAD at the time *git stash* was run, it restores the originally stashed state with no conflicts.

clear

Remove all the stash entries. Note that those entries will then be subject to pruning, and may be impossible to recover (see *Examples* below for a possible strategy).

drop [-q|--quiet] [<stash>]

Remove a single stash entry from the list of stash entries.

create

Create a stash entry (which is a regular commit object) and return its object name, without storing it anywhere in the ref namespace. This is intended to be useful for scripts. It is probably not the command you want to use; see "push" above.

store

Store a given stash created via *git stash create* (which is a dangling merge commit) in the stash ref, updating the stash reflog. This is intended to be useful for scripts. It is probably not the command you want to use; see "push" above.

OPTIONS

-a , --all

This option is only valid for *push* and *save* commands.

All ignored and untracked files are also stashed and then cleaned up with *git clean*.

-u , --include-untracked , --no-include-untracked

When used with the *push* and *save* commands, all untracked files are also stashed and then cleaned up with *git clean*.

When used with the *show* command, show the untracked files in the stash entry as part of the diff.

--only-untracked

This option is only valid for the *show* command.

Show only the untracked files in the stash entry as part of the diff.

--index

This option is only valid for *pop* and *apply* commands.

Tries to reinstate not only the working tree's changes, but also the index's ones. However, this can fail, when you have conflicts (which are stored in the index, where you therefore can no longer apply the changes as they were originally).

-k , --keep-index , --no-keep-index

This option is only valid for *push* and *save* commands.

All changes already added to the index are left intact.

-p , --patch

This option is only valid for *push* and *save* commands.

Interactively select hunks from the diff between HEAD and the working tree to be stashed. The stash entry is constructed such that its index state is the same as the index state of your repository, and its worktree contains only the changes you selected interactively. The selected changes are then rolled back from your worktree. See the Interactive Mode section of Section G.3.2, “git-add(1)” to learn how to operate the *--patch* mode.

The *--patch* option implies *--keep-index*. You can use *--no-keep-index* to override this.

-S , --staged

This option is only valid for *push* and *save* commands.

Stash only the changes that are currently staged. This is similar to basic *git commit* except the state is committed to the stash instead of current branch.

The *--patch* option has priority over this one.

--pathspec-from-file=<file>

This option is only valid for *push* command.

Pathspec is passed in *<file>* instead of commandline args. If *<file>* is exactly - then standard input is used. Pathspec elements are separated by LF or CR/LF. Pathspec elements can be quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). See also *--pathspec-file-nul* and global *--literal-pathspecs*.

--pathspec-file-nul

This option is only valid for *push* command.

Only meaningful with *--pathspec-from-file*. Pathspec elements are separated with NUL character and all other characters are taken literally (including newlines and quotes).

-q , --quiet

This option is only valid for *apply*, *drop*, *pop*, *push*, *save*, *store* commands.

Quiet, suppress feedback messages.

--

This option is only valid for *push* command.

Separates pathspec from options for disambiguation purposes.

<pathspec>…

This option is only valid for *push* command.

The new stash entry records the modified states only for the files that match the pathspec. The index entries and working tree files are then rolled back to the state in HEAD only for these files, too, leaving files that do not match the pathspec intact.

For more details, see the *pathspec* entry in Section G.4.19, “gitglossary(7)”.

<stash>

This option is only valid for *apply*, *branch*, *drop*, *pop*, *show* commands.

A reference of the form *stash@{<revision>}*. When no *<stash>* is given, the latest stash is assumed (that is, *stash@{0}*).

DISCUSSION

A stash entry is represented as a commit whose tree records the state of the working directory, and its first parent is the commit at *HEAD* when the entry was created. The tree of the second parent records the state of the index when the entry is made, and it is made a child of the *HEAD* commit. The ancestry graph looks like this:

```
        .----W
       /    /
-----H----I
```

where *H* is the *HEAD* commit, *I* is a commit that records the state of the index, and *W* is a commit that records the state of the working tree.

EXAMPLES

Pulling into a dirty tree

When you are in the middle of something, you learn that there are upstream changes that are possibly relevant to what you are doing. When your local changes do not conflict with the changes in the upstream, a simple *git pull* will let you move forward.

However, there are cases in which your local changes do conflict with the upstream changes, and *git pull* refuses to overwrite your changes. In such a case, you can stash your changes away, perform a pull, and then unstash, like this:

```
$ git pull
 ...
file foobar not up to date, cannot merge.
$ git stash
$ git pull
$ git stash pop
```

Interrupted workflow

When you are in the middle of something, your boss comes in and demands that you fix something immediately. Traditionally, you would make a commit to a temporary branch to store your changes away, and return to your original branch to make the emergency fix, like this:

```
# ... hack hack hack ...
$ git switch -c my_wip
$ git commit -a -m "WIP"
$ git switch master
$ edit emergency fix
$ git commit -a -m "Fix in a hurry"
$ git switch my_wip
$ git reset --soft HEAD^
# ... continue hacking ...
```

You can use *git stash* to simplify the above, like this:

```
# ... hack hack hack ...
$ git stash
$ edit emergency fix
$ git commit -a -m "Fix in a hurry"
$ git stash pop
# ... continue hacking ...
```

Testing partial commits

You can use *git stash push --keep-index* when you want to make two or more commits out of the changes in the work tree, and you want to test each change before committing:

```
# ... hack hack hack ...
$ git add --patch foo            # add just first part to the index
$ git stash push --keep-index    # save all other changes to the stash
$ edit/build/test first part
$ git commit -m 'First part'     # commit fully tested change
$ git stash pop                  # prepare to work on all other changes
# ... repeat above five steps until one commit remains ...
$ edit/build/test remaining parts
$ git commit foo -m 'Remaining parts'
```

Saving unrelated changes for future use

When you are in the middle of massive changes and you find some unrelated issue that you don't want to forget to fix, you can do the change(s), stage them, and use *git stash push --staged* to stash them out for future use. This is similar to committing the staged changes, only the commit ends-up being in the stash and not on the current branch.

```
# ... hack hack hack ...
$ git add --patch foo           # add unrelated changes to the index
$ git stash push --staged       # save these changes to the stash
# ... hack hack hack, finish current changes ...
$ git commit -m 'Massive'       # commit fully tested changes
$ git switch fixup-branch       # switch to another branch
$ git stash pop                 # to finish work on the saved changes
```

Recovering stash entries that were cleared/dropped erroneously

If you mistakenly drop or clear stash entries, they cannot be recovered through the normal safety mechanisms. However, you can try the following incantation to get a list of stash entries that are still in your repository, but not reachable any more:

```
git fsck --unreachable |
grep commit | cut -d\  -f3 |
xargs git log --merges --no-walk --grep=WIP
```

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

stash.showIncludeUntracked

If this is set to true, the *git stash show* command will show the untracked files of a stash entry. Defaults to false. See the description of the *show* command in Section G.3.137, “git-stash(1)”.

stash.showPatch

If this is set to true, the *git stash show* command without an option will show the stash entry in patch form. Defaults to false. See the description of the *show* command in Section G.3.137, “git-stash(1)”.

stash.showStat

If this is set to true, the *git stash show* command without an option will show a diffstat of the stash entry. Defaults to true. See the description of the *show* command in Section G.3.137, “git-stash(1)”.

SEE ALSO

Section G.3.19, “git-checkout(1)”, Section G.3.28, “git-commit(1)”, Section G.3.109, “git-reflog(1)”, Section G.3.118, “git-reset(1)”, Section G.3.140, “git-switch(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.138. git-status(1)

2024-07-02

Revision History

2024-07-02

NAME

git-status - Show the working tree status

SYNOPSIS

git status [<options>] [--] [<pathspec>…]

DESCRIPTION

Displays paths that have differences between the index file and the current HEAD commit, paths that have differences between the working tree and the index file, and paths in the working tree that are not tracked by Git (and are not ignored by Section G.4.5, “gitignore(5)”). The first are what you *would* commit by running *git commit*; the second and third are what you *could* commit by running *git add* before running *git commit*.

OPTIONS

-s , --short

Give the output in the short-format.

-b , --branch

Show the branch and tracking info even in short-format.

--show-stash

Show the number of entries currently stashed away.

--porcelain[=<version>]

Give the output in an easy-to-parse format for scripts. This is similar to the short output, but will remain stable across Git versions and regardless of user configuration. See below for details.

The version parameter is used to specify the format version. This is optional and defaults to the original version *v1* format.

--long

Give the output in the long-format. This is the default.

-v , --verbose

In addition to the names of files that have been changed, also show the textual changes that are staged to be committed (i.e., like the output of *git diff --cached*). If *-v* is specified twice, then also show the changes in the working tree that have not yet been staged (i.e., like the output of *git diff*).

-u[<mode>] , --untracked-files[=<mode>]

Show untracked files.

The mode parameter is used to specify the handling of untracked files. It is optional: it defaults to *all*, and if specified, it must be stuck to the option (e.g. *-uno*, but not *-u no*).

The possible options are:

- *no* - Show no untracked files.
- *normal* - Shows untracked files and directories.
- *all* - Also shows individual files in untracked directories.

When *-u* option is not used, untracked files and directories are shown (i.e. the same as specifying *normal*), to help you avoid forgetting to add newly created files. Because it takes extra work to find untracked files in the filesystem, this mode may take some time in a large working tree. Consider enabling untracked cache and split index if supported (see *git update-index --untracked-cache* and *git update-index --split-index*), Otherwise you can use *no* to have *git status* return more quickly without showing untracked files. All usual spellings for Boolean value *true* are taken as *normal* and *false* as *no*.

The default can be changed using the status.showUntrackedFiles configuration variable documented in Section G.3.29, “git-config(1)”.

--ignore-submodules[=<when>]

Ignore changes to submodules when looking for changes. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in Section G.3.29, “git-config(1)” or Section G.4.10, “gitmodules(5)”. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior before 1.7.0). Using "all" hides all changes to submodules (and suppresses the output of submodule summaries when the config option *status.submoduleSummary* is set).

--ignored[=<mode>]

Show ignored files as well.

The mode parameter is used to specify the handling of ignored files. It is optional: it defaults to *traditional*.

The possible options are:

- *traditional* - Shows ignored files and directories, unless --untracked-files=all is specified, in which case individual files in ignored directories are displayed.
- *no* - Show no ignored files.
- *matching* - Shows ignored files and directories matching an ignore pattern.

When *matching* mode is specified, paths that explicitly match an ignored pattern are shown. If a directory matches an ignore pattern, then it is shown, but not paths contained in the ignored directory. If a directory does not match an ignore pattern, but all contents are ignored, then the directory is not shown, but all contents are shown.

-z

Terminate entries with NUL, instead of LF. This implies the *--porcelain=v1* output format if no other format is given.

--column[=<options>] , --no-column

Display untracked files in columns. See configuration variable *column.status* for option syntax. *--column* and *--no-column* without options are equivalent to *always* and *never* respectively.

--ahead-behind , --no-ahead-behind

Display or do not display detailed ahead/behind counts for the branch relative to its upstream branch. Defaults to true.

--renames , --no-renames

Turn on/off rename detection regardless of user configuration. See also Section G.3.44, “git-diff(1)” *--no-renames*.

--find-renames[=<n>]

Turn on rename detection, optionally setting the similarity threshold. See also Section G.3.44, “git-diff(1)” *--find-renames*.

<pathspec>…

See the *pathspec* entry in Section G.4.19, “gitglossary(7)”.

OUTPUT

The output from this command is designed to be used as a commit template comment. The default, long format, is designed to be human readable, verbose and descriptive. Its contents and format are subject to change at any time.

The paths mentioned in the output, unlike many other Git commands, are made relative to the current directory if you are working in a subdirectory (this is on purpose, to help cutting and pasting). See the status.relativePaths config option below.

1. Short Format

In the short-format, the status of each path is shown as one of these forms

```
XY PATH
XY ORIG_PATH -> PATH
```

where *ORIG_PATH* is where the renamed/copied contents came from. *ORIG_PATH* is only shown when the entry is renamed or copied. The *XY* is a two-letter status code.

The fields (including the ->) are separated from each other by a single space. If a filename contains whitespace or other nonprintable characters, that field will be quoted in the manner of a C string literal: surrounded by ASCII double quote (34) characters, and with interior special characters backslash-escaped.

There are three different types of states that are shown using this format, and each one uses the *XY* syntax differently:

- When a merge is occurring and the merge was successful, or outside of a merge situation, *X* shows the status of the index and *Y* shows the status of the working tree.
- When a merge conflict has occurred and has not yet been resolved, *X* and *Y* show the state introduced by each head of the merge, relative to the common ancestor. These paths are said to be *unmerged*.
- When a path is untracked, *X* and *Y* are always the same, since they are unknown to the index. *??* is used for untracked paths. Ignored files are not listed unless --*ignored* is used; if it is, ignored files are indicated by *!!*.

Note that the term *merge* here also includes rebases using the default --*merge* strategy, cherry-picks, and anything else using the merge machinery.

In the following table, these three classes are shown in separate sections, and these characters are used for *X* and *Y* fields for the first two sections that show tracked paths:

- ' ' = unmodified
- *M* = modified
- *T* = file type changed (regular file, symbolic link or submodule)
- *A* = added
- *D* = deleted
- *R* = renamed
- *C* = copied (if config option status.renames is set to "copies")
- *U* = updated but unmerged

```
X          Y     Meaning
-------------------------------------------------
         [AMD]   not updated
M        [ MTD]  updated in index
T        [ MTD]  type changed in index
A        [ MTD]  added to index
D                deleted from index
R        [ MTD]  renamed in index
C        [ MTD]  copied in index
[MTARC]          index and work tree matches
[ MTARC]    M    work tree changed since index
[ MTARC]    T    type changed in work tree since index
[ MTARC]    D    deleted in work tree
            R    renamed in work tree
            C    copied in work tree
-------------------------------------------------
D           D    unmerged, both deleted
A           U    unmerged, added by us
U           D    unmerged, deleted by them
U           A    unmerged, added by them
D           U    unmerged, deleted by us
A           A    unmerged, both added
U           U    unmerged, both modified
```

```
-------------------------------------------------
?           ?    untracked
!           !    ignored
-------------------------------------------------
```

Submodules have more state and instead report

- *M* = the submodule has a different HEAD than recorded in the index
- *m* = the submodule has modified content
- *?* = the submodule has untracked files

This is since modified content or untracked files in a submodule cannot be added via *git add* in the superproject to prepare a commit.

m and *?* are applied recursively. For example if a nested submodule in a submodule contains an untracked file, this is reported as *?* as well.

If -b is used the short-format status is preceded by a line

```
## branchname tracking info
```

2. Porcelain Format Version 1

Version 1 porcelain format is similar to the short format, but is guaranteed not to change in a backwards-incompatible way between Git versions or based on user configuration. This makes it ideal for parsing by scripts. The description of the short format above also describes the porcelain format, with a few exceptions:

1. The user's color.status configuration is not respected; color will always be off.
2. The user's status.relativePaths configuration is not respected; paths shown will always be relative to the repository root.

There is also an alternate -z format recommended for machine parsing. In that format, the status field is the same, but some other things change. First, the -> is omitted from rename entries and the field order is reversed (e.g *from* -> *to* becomes *to from*). Second, a NUL (ASCII 0) follows each filename, replacing space as a field separator and the terminating newline (but a space still separates the status field from the first filename). Third, filenames containing special characters are not specially formatted; no quoting or backslash-escaping is performed.

Any submodule changes are reported as modified *M* instead of *m* or single *?*.

3. Porcelain Format Version 2

Version 2 format adds more detailed information about the state of the worktree and changed items. Version 2 also defines an extensible set of easy to parse optional headers.

Header lines start with "#" and are added in response to specific command line arguments. Parsers should ignore headers they don't recognize.

3.1. Branch Headers

If *--branch* is given, a series of header lines are printed with information about the current branch.

```
Line                                      Notes
------------------------------------------------------------
# branch.oid <commit> | (initial)         Current commit.
# branch.head <branch> | (detached)       Current branch.
# branch.upstream <upstream-branch>       If upstream is set.
# branch.ab +<ahead> -<behind>            If upstream is set and
                                          the commit is present.
------------------------------------------------------------
```

3.2. Stash Information

If *--show-stash* is given, one line is printed showing the number of stash entries if non-zero:

```
# stash <N>
```

3.3. Changed Tracked Entries

Following the headers, a series of lines are printed for tracked entries. One of three different line formats may be used to describe an entry depending on the type of change. Tracked entries are printed in an undefined order; parsers should allow for a mixture of the 3 line types in any order.

Ordinary changed entries have the following format:

```
1 <XY> <sub> <mH> <mI> <mW> <hH> <hI> <path>
```

Renamed or copied entries have the following format:

```
2 <XY> <sub> <mH> <mI> <mW> <hH> <hI> <X><score> <path><sep><origPath>

Field       Meaning
--------------------------------------------------------
<XY>        A 2 character field containing the staged and
            unstaged XY values described in the short format,
            with unchanged indicated by a "." rather than
            a space.
<sub>       A 4 character field describing the submodule state.
            "N..." when the entry is not a submodule.
            "S<c><m><u>" when the entry is a submodule.
            <c> is "C" if the commit changed; otherwise ".".
            <m> is "M" if it has tracked changes; otherwise ".".
            <u> is "U" if there are untracked changes; otherwise ".".
<mH>        The octal file mode in HEAD.
<mI>        The octal file mode in the index.
<mW>        The octal file mode in the worktree.
<hH>        The object name in HEAD.
<hI>        The object name in the index.
<X><score>  The rename or copy score (denoting the percentage
            of similarity between the source and target of the
            move or copy). For example "R100" or "C75".
<path>      The pathname.  In a renamed/copied entry, this
            is the target path.
<sep>       When the `-z` option is used, the 2 pathnames are separated
            with a NUL (ASCII 0x00) byte; otherwise, a tab (ASCII 0x09)
            byte separates them.
<origPath>  The pathname in the commit at HEAD or in the index.
            This is only present in a renamed/copied entry, and
            tells where the renamed/copied contents came from.
--------------------------------------------------------
```

Unmerged entries have the following format; the first character is a "u" to distinguish from ordinary changed entries.

```
u <XY> <sub> <m1> <m2> <m3> <mW> <h1> <h2> <h3> <path>

Field       Meaning
--------------------------------------------------------
<XY>        A 2 character field describing the conflict type
            as described in the short format.
<sub>       A 4 character field describing the submodule state
```

```
            as described above.
<m1>        The octal file mode in stage 1.
<m2>        The octal file mode in stage 2.
<m3>        The octal file mode in stage 3.
<mW>        The octal file mode in the worktree.
<h1>        The object name in stage 1.
<h2>        The object name in stage 2.
<h3>        The object name in stage 3.
<path>      The pathname.
-------------------------------------------------------
```

3.4. Other Items

Following the tracked entries (and if requested), a series of lines will be printed for untracked and then ignored items found in the worktree.

Untracked items have the following format:

```
? <path>
```

Ignored items have the following format:

```
! <path>
```

3.5. Pathname Format Notes and -z

When the *-z* option is given, pathnames are printed as is and without any quoting and lines are terminated with a NUL (ASCII 0x00) byte.

Without the *-z* option, pathnames with "unusual" characters are quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”).

CONFIGURATION

The command honors *color.status* (or *status.color* -- they mean the same thing and the latter is kept for backward compatibility) and *color.status.<slot>* configuration variables to colorize its output.

If the config variable *status.relativePaths* is set to false, then all paths shown are relative to the repository root, not to the current directory.

If *status.submoduleSummary* is set to a non zero number or true (identical to -1 or an unlimited number), the submodule summary will be enabled for the long format and a summary of commits for modified submodules will be shown (see --summary-limit option of Section G.3.141, “git-submodule(1)”). Please note that the summary output from the status command will be suppressed for all submodules when *diff.ignoreSubmodules* is set to *all* or only for those submodules where *submodule.<name>.ignore=all*. To also view the summary for ignored submodules you can either use the --ignore-submodules=dirty command line option or the *git submodule summary* command, which shows a similar output but does not honor these settings.

BACKGROUND REFRESH

By default, *git status* will automatically refresh the index, updating the cached stat information from the working tree and writing out the result. Writing out the updated index is an optimization that isn't strictly necessary (*status* computes the values for itself, but writing them out is just to save subsequent programs from repeating our computation). When *status* is run in the background, the lock held during the write may conflict with other simultaneous processes, causing them to fail. Scripts running *status* in the background should consider using *git --no-optional-locks status* (see Section G.3.1, “git(1)” for details).

UNTRACKED FILES AND PERFORMANCE

git status can be very slow in large worktrees if/when it needs to search for untracked files and directories. There are many configuration options available to speed this up by either avoiding the work or making use of cached

results from previous Git commands. There is no single optimum set of settings right for everyone. We'll list a summary of the relevant options to help you, but before going into the list, you may want to run *git status* again, because your configuration may already be caching *git status* results, so it could be faster on subsequent runs.

- The *--untracked-files=no* flag or the *status.showUntrackedFiles=no* config (see above for both): indicate that *git status* should not report untracked files. This is the fastest option. *git status* will not list the untracked files, so you need to be careful to remember if you create any new files and manually *git add* them.
- *advice.statusUoption=false* (see Section G.3.29, "git-config(1)"): setting this variable to *false* disables the warning message given when enumerating untracked files takes more than 2 seconds. In a large project, it may take longer and the user may have already accepted the trade off (e.g. using "-uno" may not be an acceptable option for the user), in which case, there is no point issuing the warning message, and in such a case, disabling the warning may be the best.
- *core.untrackedCache=true* (see Section G.3.147, "git-update-index(1)"): enable the untracked cache feature and only search directories that have been modified since the previous *git status* command. Git remembers the set of untracked files within each directory and assumes that if a directory has not been modified, then the set of untracked files within has not changed. This is much faster than enumerating the contents of every directory, but still not without cost, because Git still has to search for the set of modified directories. The untracked cache is stored in the *.git/index* file. The reduced cost of searching for untracked files is offset slightly by the increased size of the index and the cost of keeping it up-to-date. That reduced search time is usually worth the additional size.
- *core.untrackedCache=true* and *core.fsmonitor=true* or *core.fsmonitor=<hook-command-pathname>* (see Section G.3.147, "git-update-index(1)"): enable both the untracked cache and FSMonitor features and only search directories that have been modified since the previous *git status* command. This is faster than using just the untracked cache alone because Git can also avoid searching for modified directories. Git only has to enumerate the exact set of directories that have changed recently. While the FSMonitor feature can be enabled without the untracked cache, the benefits are greatly reduced in that case.

Note that after you turn on the untracked cache and/or FSMonitor features it may take a few *git status* commands for the various caches to warm up before you see improved command times. This is normal.

SEE ALSO

Section G.4.5, "gitignore(5)"

GIT

Part of the Section G.3.1, "git(1)" suite

G.3.139. git-stripspace(1)

2024-07-02
Revision History
2024-07-02

NAME

git-stripspace - Remove unnecessary whitespace

SYNOPSIS

git stripspace [-s | --strip-comments]
git stripspace [-c | --comment-lines]

DESCRIPTION

Read text, such as commit messages, notes, tags and branch descriptions, from the standard input and clean it in the manner used by Git.

With no arguments, this will:

- remove trailing whitespace from all lines
- collapse multiple consecutive empty lines into one empty line
- remove empty lines from the beginning and end of the input
- add a missing *\n* to the last line if necessary.

In the case where the input consists entirely of whitespace characters, no output will be produced.

NOTE: This is intended for cleaning metadata. Prefer the *--whitespace=fix* mode of Section G.3.5, "git-apply(1)" for correcting whitespace of patches or files in the repository.

OPTIONS

-s , --strip-comments

Skip and remove all lines starting with a comment character (default *#*).

-c , --comment-lines

Prepend the comment character and a blank space to each line. Lines will automatically be terminated with a newline. On empty lines, only the comment character will be prepended.

EXAMPLES

Given the following noisy input with *$* indicating the end of a line:

```
|A brief introduction   $
|   $
|$
|A new paragraph$
|# with a commented-out line    $
|explaining lots of stuff.$
|$
|# An old paragraph, also commented-out. $
|      $
|The end.$
|  $
```

Use *git stripspace* with no arguments to obtain:

```
|A brief introduction$
|$
|A new paragraph$
|# with a commented-out line$
|explaining lots of stuff.$
|$
|# An old paragraph, also commented-out.$
|$
|The end.$
```

Use *git stripspace --strip-comments* to obtain:

```
|A brief introduction$
|$
|A new paragraph$
|explaining lots of stuff.$
|$
|The end.$
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.140. git-switch(1)

2024-07-02
Revision History
2024-07-02

NAME

git-switch - Switch branches

SYNOPSIS

git switch [<options>] [--no-guess] <branch>
git switch [<options>] --detach [<start-point>]
git switch [<options>] (-c|-C) <new-branch> [<start-point>]
git switch [<options>] --orphan <new-branch>

DESCRIPTION

Switch to a specified branch. The working tree and the index are updated to match the branch. All new commits will be added to the tip of this branch.

Optionally a new branch could be created with either *-c*, *-C*, automatically from a remote branch of same name (see *--guess*), or detach the working tree from any branch with *--detach*, along with switching.

Switching branches does not require a clean index and working tree (i.e. no differences compared to *HEAD*). The operation is aborted however if the operation leads to loss of local changes, unless told otherwise with *--discard-changes* or *--merge*.

THIS COMMAND IS EXPERIMENTAL. THE BEHAVIOR MAY CHANGE.

OPTIONS

<branch>

Branch to switch to.

<new-branch>

Name for the new branch.

<start-point>

The starting point for the new branch. Specifying a *<start-point>* allows you to create a branch based on some other point in history than where HEAD currently points. (Or, in the case of *--detach*, allows you to inspect and detach from some other point.)

You can use the *@{-N}* syntax to refer to the N-th last branch/commit switched to using "git switch" or "git checkout" operation. You may also specify - which is synonymous to *@{-1}*. This is often used to switch quickly between two branches, or to undo a branch switch by mistake.

As a special case, you may use *A...B* as a shortcut for the merge base of *A* and *B* if there is exactly one merge base. You can leave out at most one of *A* and *B*, in which case it defaults to *HEAD*.

-c <new-branch> , --create <new-branch>

Create a new branch named *<new-branch>* starting at *<start-point>* before switching to the branch. This is the transactional equivalent of

```
$ git branch <new-branch>
$ git switch <new-branch>
```

that is to say, the branch is not reset/created unless "git switch" is successful (e.g., when the branch is in use in another worktree, not just the current branch stays the same, but the branch is not reset to the start-point, either).

-C <new-branch> , --force-create <new-branch>

Similar to *--create* except that if *<new-branch>* already exists, it will be reset to *<start-point>*. This is a convenient shortcut for:

```
$ git branch -f <new-branch>
$ git switch <new-branch>
```

-d , --detach

Switch to a commit for inspection and discardable experiments. See the "DETACHED HEAD" section in Section G.3.19, “git-checkout(1)” for details.

--guess , --no-guess

If *<branch>* is not found but there does exist a tracking branch in exactly one remote (call it *<remote>*) with a matching name, treat as equivalent to

```
$ git switch -c <branch> --track <remote>/<branch>
```

If the branch exists in multiple remotes and one of them is named by the *checkout.defaultRemote* configuration variable, we'll use that one for the purposes of disambiguation, even if the *<branch>* isn't unique across all remotes. Set it to e.g. *checkout.defaultRemote=origin* to always checkout remote branches from there if *<branch>* is ambiguous but exists on the *origin* remote. See also *checkout.defaultRemote* in Section G.3.29, “git-config(1)”.

--guess is the default behavior. Use *--no-guess* to disable it.

The default behavior can be set via the *checkout.guess* configuration variable.

-f , --force

An alias for *--discard-changes*.

--discard-changes

Proceed even if the index or the working tree differs from *HEAD*. Both the index and working tree are restored to match the switching target. If *--recurse-submodules* is specified, submodule content is also restored to match the switching target. This is used to throw away local changes.

-m , --merge

If you have local modifications to one or more files that are different between the current branch and the branch to which you are switching, the command refuses to switch branches in order to preserve your modifications in context. However, with this option, a three-way merge between the current branch, your working tree contents, and the new branch is done, and you will be on the new branch.

When a merge conflict happens, the index entries for conflicting paths are left unmerged, and you need to resolve the conflicts and mark the resolved paths with *git add* (or *git rm* if the merge should result in deletion of the path).

--conflict=<style>

The same as *--merge* option above, but changes the way the conflicting hunks are presented, overriding the *merge.conflictStyle* configuration variable. Possible values are "merge" (default), "diff3", and "zdiff3".

-q , --quiet

Quiet, suppress feedback messages.

--progress , --no-progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless *--quiet* is specified. This flag enables progress reporting even if not attached to a terminal, regardless of *--quiet*.

-t , --track [direct|inherit]

When creating a new branch, set up "upstream" configuration. *-c* is implied. See *--track* in Section G.3.10, “git-branch(1)” for details.

If no *-c* option is given, the name of the new branch will be derived from the remote-tracking branch, by looking at the local part of the refspec configured for the corresponding remote, and then stripping the initial part up to the "*". This would tell us to use *hack* as the local branch when branching off of *origin/hack* (or *remotes/origin/hack*, or even *refs/remotes/origin/hack*). If the given name has no slash, or the above guessing results in an empty name, the guessing is aborted. You can explicitly give a name with *-c* in such a case.

--no-track

Do not set up "upstream" configuration, even if the *branch.autoSetupMerge* configuration variable is true.

--orphan <new-branch>

Create a new unborn branch, named <*new-branch*>. All tracked files are removed.

--ignore-other-worktrees

git switch refuses when the wanted ref is already checked out by another worktree. This option makes it check the ref out anyway. In other words, the ref can be held by more than one worktree.

--recurse-submodules , --no-recurse-submodules

Using *--recurse-submodules* will update the content of all active submodules according to the commit recorded in the superproject. If nothing (or *--no-recurse-submodules*) is used, submodules working trees will not be updated. Just like Section G.3.141, “git-submodule(1)”, this will detach *HEAD* of the submodules.

EXAMPLES

The following command switches to the "master" branch:

```
$ git switch master
```

After working in the wrong branch, switching to the correct branch would be done using:

```
$ git switch mytopic
```

However, your "wrong" branch and correct "mytopic" branch may differ in files that you have modified locally, in which case the above switch would fail like this:

```
$ git switch mytopic
error: You have local changes to 'frotz'; not switching branches.
```

You can give the *-m* flag to the command, which would try a three-way merge:

```
$ git switch -m mytopic
Auto-merging frotz
```

After this three-way merge, the local modifications are *not* registered in your index file, so *git diff* would show you what changes you made since the tip of the new branch.

To switch back to the previous branch before we switched to mytopic (i.e. "master" branch):

```
$ git switch -
```

You can grow a new branch from any commit. For example, switch to "HEAD~3" and create branch "fixup":

```
$ git switch -c fixup HEAD~3
Switched to a new branch 'fixup'
```

If you want to start a new branch from a remote branch of the same name:

```
$ git switch new-topic
Branch 'new-topic' set up to track remote branch 'new-topic' from 'origin'
Switched to a new branch 'new-topic'
```

To check out commit *HEAD~3* for temporary inspection or experiment without creating a new branch:

```
$ git switch --detach HEAD~3
HEAD is now at 9fc9555312 Merge branch 'cc/shared-index-permbits'
```

If it turns out whatever you have done is worth keeping, you can always create a new name for it (without switching away):

```
$ git switch -c good-surprises
```

CONFIGURATION

Everything below this line in this section is selectively included from the Section G.3.29, “git-config(1)” documentation. The content is the same as what's found there:

checkout.defaultRemote

When you run *git checkout* <*something*> or *git switch* <*something*> and only have one remote, it may implicitly fall back on checking out and tracking e.g. *origin/*<*something*>. This stops working as soon as you have more than one remote with a <*something*> reference. This setting allows for setting the name of a preferred remote that should always win when it comes to disambiguation. The typical use-case is to set this to *origin*.

Currently this is used by Section G.3.140, “git-switch(1)” and Section G.3.19, “git-checkout(1)” when *git checkout* <*something*> or *git switch* <*something*> will checkout the <*something*> branch on another remote, and by Section G.3.159, “git-worktree(1)” when *git worktree add* refers to a remote branch. This setting might be used for other checkout-like commands or functionality in the future.

checkout.guess

Provides the default value for the *--guess* or *--no-guess* option in *git checkout* and *git switch*. See Section G.3.140, “git-switch(1)” and Section G.3.19, “git-checkout(1)”.

checkout.workers

The number of parallel workers to use when updating the working tree. The default is one, i.e. sequential execution. If set to a value less than one, Git will use as many workers as the number of logical cores available. This setting and *checkout.thresholdForParallelism* affect all commands that perform checkout. E.g. checkout, clone, reset, sparse-checkout, etc.

Note: Parallel checkout usually delivers better performance for repositories located on SSDs or over NFS. For repositories on spinning disks and/or machines with a small number of cores, the default sequential checkout often performs better. The size and compression level of a repository might also influence how well the parallel version performs.

checkout.thresholdForParallelism

When running parallel checkout with a small number of files, the cost of subprocess spawning and inter-process communication might outweigh the parallelization gains. This setting allows you to define the minimum number of files for which parallel checkout should be attempted. The default is 100.

SEE ALSO

Section G.3.19, “git-checkout(1)”, Section G.3.10, “git-branch(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.141. git-submodule(1)

2024-07-02
Revision History
2024-07-02

NAME

git-submodule - Initialize, update or inspect submodules

SYNOPSIS

git submodule [--quiet] [--cached]
git submodule [--quiet] add [<options>] [--] <repository> [<path>]
git submodule [--quiet] status [--cached] [--recursive] [--] [<path>…]
git submodule [--quiet] init [--] [<path>…]
git submodule [--quiet] deinit [-f|--force] (--all|[--] <path>…)
git submodule [--quiet] update [<options>] [--] [<path>…]
git submodule [--quiet] set-branch [<options>] [--] <path>
git submodule [--quiet] set-url [--] <path> <newurl>
git submodule [--quiet] summary [<options>] [--] [<path>…]
git submodule [--quiet] foreach [--recursive] <command>
git submodule [--quiet] sync [--recursive] [--] [<path>…]
git submodule [--quiet] absorbgitdirs [--] [<path>…]

DESCRIPTION

Inspects, updates and manages submodules.

For more information about submodules, see Section G.4.15, “gitsubmodules(7)”.

COMMANDS

With no arguments, shows the status of existing submodules. Several subcommands are available to perform operations on the submodules.

add [-b <branch>] [-f|--force] [--name <name>] [--reference <repository>] [--depth <depth>] [--] <repository> [<path>]

Add the given repository as a submodule at the given path to the changeset to be committed next to the current project: the current project is termed the "superproject".

<repository> is the URL of the new submodule's origin repository. This may be either an absolute URL, or (if it begins with ./ or ../), the location relative to the superproject's default remote repository (Please note that to specify a repository *foo.git* which is located right next to a superproject *bar.git*, you'll have to use *../foo.git* instead of *./foo.git* - as one might expect when following the rules for relative URLs - because the evaluation of relative URLs in Git is identical to that of relative directories).

The default remote is the remote of the remote-tracking branch of the current branch. If no such remote-tracking branch exists or the HEAD is detached, "origin" is assumed to be the default remote. If the superproject doesn't have a default remote configured the superproject is its own authoritative upstream and the current working directory is used instead.

The optional argument <path> is the relative location for the cloned submodule to exist in the superproject. If <path> is not given, the canonical part of the source repository is used ("repo" for "/path/to/repo.git" and "foo" for "host.xz:foo/.git"). If <path> exists and is already a valid Git repository, then it is staged for commit without cloning. The <path> is also used as the submodule's logical name in its configuration entries unless *--name* is used to specify a logical name.

The given URL is recorded into *.gitmodules* for use by subsequent users cloning the superproject. If the URL is given relative to the superproject's repository, the presumption is the superproject and submodule repositories will be kept together in the same relative location, and only the superproject's URL needs to be provided. git-submodule will correctly locate the submodule using the relative URL in *.gitmodules*.

status [--cached] [--recursive] [--] [<path>…]

Show the status of the submodules. This will print the SHA-1 of the currently checked out commit for each submodule, along with the submodule path and the output of *git describe* for the SHA-1. Each SHA-1 will possibly be prefixed with - if the submodule is not initialized, + if the currently checked out submodule commit does not match the SHA-1 found in the index of the containing repository and *U* if the submodule has merge conflicts.

If *--cached* is specified, this command will instead print the SHA-1 recorded in the superproject for each submodule.

If *--recursive* is specified, this command will recurse into nested submodules, and show their status as well.

If you are only interested in changes of the currently initialized submodules with respect to the commit recorded in the index or the HEAD, Section G.3.138, “git-status(1)” and Section G.3.44, “git-diff(1)” will provide that information too (and can also report changes to a submodule's work tree).

init [--] [<path>…]

Initialize the submodules recorded in the index (which were added and committed elsewhere) by setting *submodule.$name.url* in *.git/config*, using the same setting from *.gitmodules* as a template. If the URL is relative, it will be resolved using the default remote. If there is no default remote, the current repository will be assumed to be upstream.

Optional <path> arguments limit which submodules will be initialized. If no path is specified and submodule.active has been configured, submodules configured to be active will be initialized, otherwise all submodules are initialized.

It will also copy the value of *submodule.$name.update*, if present in the *.gitmodules* file, to *.git/config*, but (1) this command does not alter existing information in *.git/config*, and (2) *submodule.$name.update* that is set to a custom command is **not** copied for security reasons.

You can then customize the submodule clone URLs in *.git/config* for your local setup and proceed to *git submodule update*; you can also just use *git submodule update --init* without the explicit *init* step if you do not intend to customize any submodule locations.

See the add subcommand for the definition of default remote.

deinit [-f|--force] (--all|[--] <path>…)

Unregister the given submodules, i.e. remove the whole *submodule.$name* section from .git/config together with their work tree. Further calls to *git submodule update*, *git submodule foreach* and *git submodule sync* will skip any unregistered submodules until they are initialized again, so use this command if you don't want to have a local checkout of the submodule in your working tree anymore.

When the command is run without pathspec, it errors out, instead of deinit-ing everything, to prevent mistakes.

If *--force* is specified, the submodule's working tree will be removed even if it contains local modifications.

If you really want to remove a submodule from the repository and commit that use Section G.3.123, “git-rm(1)” instead. See Section G.4.15, “gitsubmodules(7)” for removal options.

update [--init] [--remote] [-N|--no-fetch] [--[no-]recommend-shallow] [-f|--force] [--checkout|--rebase|--merge] [--reference <repository>] [--depth <depth>] [--recursive] [--jobs <n>] [--[no-]single-branch] [--filter <filter-spec>] [--] [<path>…]

Update the registered submodules to match what the superproject expects by cloning missing submodules, fetching missing commits in submodules and updating the working tree of the submodules. The "updating" can be done in several ways depending on command line options and the value of *submodule.<name>.update* configuration variable. The command line option takes precedence over the configuration variable. If neither is given, a *checkout* is performed. (note: what is in *.gitmodules* file is irrelevant at this point; see *git submodule init* above for how *.gitmodules* is used). The *update* procedures supported both from the command line as well as through the *submodule.<name>.update* configuration are:

checkout

the commit recorded in the superproject will be checked out in the submodule on a detached HEAD.

If *--force* is specified, the submodule will be checked out (using *git checkout --force*), even if the commit specified in the index of the containing repository already matches the commit checked out in the submodule.

rebase

the current branch of the submodule will be rebased onto the commit recorded in the superproject.

merge

the commit recorded in the superproject will be merged into the current branch in the submodule.

The following update procedures have additional limitations:

custom command

mechanism for running arbitrary commands with the commit ID as an argument. Specifically, if the *submodule.<name>.update* configuration variable is set to *!custom command*, the object name of the commit recorded in the superproject for the submodule is appended to the *custom command* string and executed. Note that this mechanism is not supported in the *.gitmodules* file or on the command line.

none

the submodule is not updated. This update procedure is not allowed on the command line.

If the submodule is not yet initialized, and you just want to use the setting as stored in *.gitmodules*, you can automatically initialize the submodule with the *--init* option.

If *--recursive* is specified, this command will recurse into the registered submodules, and update any nested submodules within.

If *--filter <filter-spec>* is specified, the given partial clone filter will be applied to the submodule. See Section G.3.120, “git-rev-list(1)” for details on filter specifications.

set-branch (-b|--branch) <branch> [--] <path> , set-branch (-d|--default) [--] <path>

Sets the default remote tracking branch for the submodule. The *--branch* option allows the remote branch to be specified. The *--default* option removes the submodule.<name>.branch configuration key, which causes the tracking branch to default to the remote *HEAD*.

set-url [--] <path> <newurl>

Sets the URL of the specified submodule to <newurl>. Then, it will automatically synchronize the submodule's new remote URL configuration.

summary [--cached|--files] [(-n|--summary-limit) <n>] [commit] [--] [<path>…]

Show commit summary between the given commit (defaults to HEAD) and working tree/index. For a submodule in question, a series of commits in the submodule between the given super project commit and the index or working tree (switched by *--cached*) are shown. If the option *--files* is given, show the series of commits in the submodule between the index of the super project and the working tree of the submodule (this option doesn't allow to use the *--cached* option or to provide an explicit commit).

Using the *--submodule=log* option with Section G.3.44, "git-diff(1)" will provide that information too.

foreach [--recursive] <command>

Evaluates an arbitrary shell command in each checked out submodule. The command has access to the variables $name, $sm_path, $displaypath, $sha1 and $toplevel: $name is the name of the relevant submodule section in *.gitmodules*, $sm_path is the path of the submodule as recorded in the immediate superproject, $displaypath contains the relative path from the current working directory to the submodules root directory, $sha1 is the commit as recorded in the immediate superproject, and $toplevel is the absolute path to the top-level of the immediate superproject. Note that to avoid conflicts with *$PATH* on Windows, the *$path* variable is now a deprecated synonym of *$sm_path* variable. Any submodules defined in the superproject but not checked out are ignored by this command. Unless given *--quiet*, foreach prints the name of each submodule before evaluating the command. If *--recursive* is given, submodules are traversed recursively (i.e. the given shell command is evaluated in nested submodules as well). A non-zero return from the command in any submodule causes the processing to terminate. This can be overridden by adding || *:* to the end of the command.

As an example, the command below will show the path and currently checked out commit for each submodule:

```
git submodule foreach 'echo $sm_path `git rev-parse HEAD`'
```

sync [--recursive] [--] [<path>…]

Synchronizes submodules' remote URL configuration setting to the value specified in *.gitmodules*. It will only affect those submodules which already have a URL entry in .git/config (that is the case when they are initialized or freshly added). This is useful when submodule URLs change upstream and you need to update your local repositories accordingly.

git submodule sync synchronizes all submodules while *git submodule sync -- A* synchronizes submodule "A" only.

If *--recursive* is specified, this command will recurse into the registered submodules, and sync any nested submodules within.

absorbgitdirs

If a git directory of a submodule is inside the submodule, move the git directory of the submodule into its superproject's *$GIT_DIR/modules* path and then connect the git directory and its working directory by setting the *core.worktree* and adding a .git file pointing to the git directory embedded in the superprojects git directory.

A repository that was cloned independently and later added as a submodule or old setups have the submodules git directory inside the submodule instead of embedded into the superprojects git directory.

This command is recursive by default.

OPTIONS

-q , --quiet

Only print error messages.

--progress

This option is only valid for add and update commands. Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

--all

This option is only valid for the deinit command. Unregister all submodules in the working tree.

-b <branch> , --branch <branch>

Branch of repository to add as submodule. The name of the branch is recorded as *submodule.<name>.branch* in *.gitmodules* for *update --remote*. A special value of . is used to indicate that the name of the branch in the submodule should be the same name as the current branch in the current repository. If the option is not specified, it defaults to the remote *HEAD*.

-f , --force

This option is only valid for add, deinit and update commands. When running add, allow adding an otherwise ignored submodule path. When running deinit the submodule working trees will be removed even if they contain local changes. When running update (only effective with the checkout procedure), throw away local changes in submodules when switching to a different commit; and always run a checkout operation in the submodule, even if the commit listed in the index of the containing repository matches the commit checked out in the submodule.

--cached

This option is only valid for status and summary commands. These commands typically use the commit found in the submodule HEAD, but with this option, the commit stored in the index is used instead.

--files

This option is only valid for the summary command. This command compares the commit in the index with that in the submodule HEAD when this option is used.

-n , --summary-limit

This option is only valid for the summary command. Limit the summary size (number of commits shown in total). Giving 0 will disable the summary; a negative number means unlimited (the default). This limit only applies to modified submodules. The size is always limited to 1 for added/deleted/typechanged submodules.

--remote

This option is only valid for the update command. Instead of using the superproject's recorded SHA-1 to update the submodule, use the status of the submodule's remote-tracking branch. The remote used is branch's remote (*branch.<name>.remote*), defaulting to *origin*. The remote branch used defaults to the remote *HEAD*, but the branch name may be overridden by setting the *submodule.<name>.branch* option in either *.gitmodules* or *.git/config* (with *.git/config* taking precedence).

This works for any of the supported update procedures (*--checkout*, *--rebase*, etc.). The only change is the source of the target SHA-1. For example, *submodule update --remote --merge* will merge upstream submodule changes into the submodules, while *submodule update --merge* will merge superproject gitlink changes into the submodules.

In order to ensure a current tracking branch state, *update --remote* fetches the submodule's remote repository before calculating the SHA-1. If you don't want to fetch, you should use *submodule update --remote --no-fetch*.

Use this option to integrate changes from the upstream subproject with your submodule's current HEAD. Alternatively, you can run *git pull* from the submodule, which is equivalent except for the remote branch name: *update --remote* uses the default upstream repository and *submodule.<name>.branch*, while *git pull* uses the submodule's *branch.<name>.merge*. Prefer *submodule.<name>.branch* if you want to distribute the default upstream branch with the superproject and *branch.<name>.merge* if you want a more native feel while working in the submodule itself.

-N , --no-fetch

This option is only valid for the update command. Don't fetch new objects from the remote site.

--checkout

This option is only valid for the update command. Checkout the commit recorded in the superproject on a detached HEAD in the submodule. This is the default behavior, the main use of this option is to override *submodule.$name.update* when set to a value other than *checkout*. If the key *submodule.$name.update* is either not explicitly set or set to *checkout*, this option is implicit.

--merge

This option is only valid for the update command. Merge the commit recorded in the superproject into the current branch of the submodule. If this option is given, the submodule's HEAD will not be detached. If a merge failure prevents this process, you will have to resolve the resulting conflicts within the submodule with the usual conflict resolution tools. If the key *submodule.$name.update* is set to *merge*, this option is implicit.

--rebase

This option is only valid for the update command. Rebase the current branch onto the commit recorded in the superproject. If this option is given, the submodule's HEAD will not be detached. If a merge failure prevents this process, you will have to resolve these failures with Section G.3.107, “git-rebase(1)”. If the key *submodule.$name.update* is set to *rebase*, this option is implicit.

--init

This option is only valid for the update command. Initialize all submodules for which "git submodule init" has not been called so far before updating.

--name

This option is only valid for the add command. It sets the submodule's name to the given string instead of defaulting to its path. The name must be valid as a directory name and may not end with a /.

--reference <repository>

This option is only valid for add and update commands. These commands sometimes need to clone a remote repository. In this case, this option will be passed to the Section G.3.24, “git-clone(1)” command.

NOTE: Do **not** use this option unless you have read the note for Section G.3.24, “git-clone(1)”'s *--reference*, *--shared*, and *--dissociate* options carefully.

--dissociate

This option is only valid for add and update commands. These commands sometimes need to clone a remote repository. In this case, this option will be passed to the Section G.3.24, “git-clone(1)” command.

NOTE: see the NOTE for the *--reference* option.

--recursive

This option is only valid for foreach, update, status and sync commands. Traverse submodules recursively. The operation is performed not only in the submodules of the current repo, but also in any nested submodules inside those submodules (and so on).

--depth

This option is valid for add and update commands. Create a *shallow* clone with a history truncated to the specified number of revisions. See Section G.3.24, “git-clone(1)”

--[no-]recommend-shallow

This option is only valid for the update command. The initial clone of a submodule will use the recommended *submodule.<name>.shallow* as provided by the *.gitmodules* file by default. To ignore the suggestions use *--no-recommend-shallow*.

-j <n> , --jobs <n>

This option is only valid for the update command. Clone new submodules in parallel with as many jobs. Defaults to the *submodule.fetchJobs* option.

--[no-]single-branch

This option is only valid for the update command. Clone only one branch during update: HEAD or one specified by --branch.

<path>…

Paths to submodule(s). When specified this will restrict the command to only operate on the submodules found at the specified paths. (This argument is required with add).

FILES

When initializing submodules, a *.gitmodules* file in the top-level directory of the containing repository is used to find the url of each submodule. This file should be formatted in the same way as *$GIT_DIR/config*. The key to each submodule url is "submodule.$name.url". See Section G.4.10, “gitmodules(5)” for details.

SEE ALSO

Section G.4.15, “gitsubmodules(7)”, Section G.4.10, “gitmodules(5)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.142. git-svn(1)

2024-07-02

Revision History

2024-07-02

NAME

git-svn - Bidirectional operation between a Subversion repository and Git

SYNOPSIS

git svn <command> [<options>] [<arguments>]

DESCRIPTION

git svn is a simple conduit for changesets between Subversion and Git. It provides a bidirectional flow of changes between a Subversion and a Git repository.

git svn can track a standard Subversion repository, following the common "trunk/branches/tags" layout, with the --stdlayout option. It can also follow branches and tags in any layout with the -T/-t/-b options (see options to *init* below, and also the *clone* command).

Once tracking a Subversion repository (with any of the above methods), the Git repository can be updated from Subversion by the *fetch* command and Subversion updated from Git by the *dcommit* command.

COMMANDS

init

Initializes an empty Git repository with additional metadata directories for *git svn*. The Subversion URL may be specified as a command-line argument, or as full URL arguments to -T/-t/-b. Optionally, the target directory to operate on can be specified as a second argument. Normally this command initializes the current directory.

-T<trunk-subdir> , --trunk=<trunk-subdir> , -t<tags-subdir> , --tags=<tags-subdir> , -b<branches-subdir> , --branches=<branches-subdir> , -s , --stdlayout

These are optional command-line options for init. Each of these flags can point to a relative repository path (--tags=project/tags) or a full url (--tags=https://foo.org/project/tags). You can specify more than one --tags and/or --branches options, in case your Subversion repository places tags or branches under multiple paths. The option --stdlayout is a shorthand way of setting trunk,tags,branches as the relative paths, which is the Subversion default. If any of the other options are given as well, they take precedence.

--no-metadata

Set the *noMetadata* option in the [svn-remote] config. This option is not recommended, please read the *svn.noMetadata* section of this manpage before using this option.

--use-svm-props

Set the *useSvmProps* option in the [svn-remote] config.

--use-svnsync-props

Set the *useSvnsyncProps* option in the [svn-remote] config.

--rewrite-root=<URL>

Set the *rewriteRoot* option in the [svn-remote] config.

--rewrite-uuid=<UUID>

Set the *rewriteUUID* option in the [svn-remote] config.

--username=<user>

For transports that SVN handles authentication for (http, https, and plain svn), specify the username. For other transports (e.g. *svn+ssh://*), you must include the username in the URL, e.g. *svn+ssh://foo@svn.bar.com/project*

--prefix=<prefix>

This allows one to specify a prefix which is prepended to the names of remotes if trunk/branches/tags are specified. The prefix does not automatically include a trailing slash, so be sure you include one in the argument if that is what you want. If --branches/-b is specified, the prefix must include a trailing slash. Setting a prefix (with a trailing slash) is strongly encouraged in any case, as your SVN-tracking refs will then be located at "refs/remotes/$prefix/**", which is compatible with Git's own remote-tracking ref layout (refs/remotes/$remote/**). Setting a prefix is also useful if you wish to track multiple projects that share a common repository. By default, the prefix is set to *origin/*.

> **Note**
>
> Before Git v2.0, the default prefix was "" (no prefix). This meant that SVN-tracking refs were put at "refs/remotes/*", which is incompatible with how Git's own remote-tracking refs are organized. If you still want the old default, you can get it by passing *--prefix ""* on the command line (*--prefix=""* may not work if your Perl's Getopt::Long is < v2.37).

--ignore-refs=<regex>

When passed to *init* or *clone* this regular expression will be preserved as a config key. See *fetch* for a description of *--ignore-refs*.

--ignore-paths=<regex>

When passed to *init* or *clone* this regular expression will be preserved as a config key. See *fetch* for a description of *--ignore-paths*.

--include-paths=<regex>

When passed to *init* or *clone* this regular expression will be preserved as a config key. See *fetch* for a description of *--include-paths*.

--no-minimize-url

When tracking multiple directories (using --stdlayout, --branches, or --tags options), git svn will attempt to connect to the root (or highest allowed level) of the Subversion repository. This default allows better tracking of history if entire projects are moved within a repository, but may cause issues on repositories where read access restrictions are in place. Passing *--no-minimize-url* will allow git svn to accept URLs as-is without attempting to connect to a higher level directory. This option is off by default when only one URL/branch is tracked (it would do little good).

fetch

Fetch unfetched revisions from the Subversion remote we are tracking. The name of the [svn-remote "…"] section in the $GIT_DIR/config file may be specified as an optional command-line argument.

This automatically updates the rev_map if needed (see *$GIT_DIR/svn/**/.rev_map.** in the FILES section below for details).

--localtime

Store Git commit times in the local time zone instead of UTC. This makes *git log* (even without --date=local) show the same times that *svn log* would in the local time zone.

This doesn't interfere with interoperating with the Subversion repository you cloned from, but if you wish for your local Git repository to be able to interoperate with someone else's local Git repository, either don't use this option or you should both use it in the same local time zone.

--parent

Fetch only from the SVN parent of the current HEAD.

--ignore-refs=<regex>

Ignore refs for branches or tags matching the Perl regular expression. A "negative look-ahead assertion" like *^refs/remotes/origin/(?!tags/wanted-tag|wanted-branch).*$* can be used to allow only certain refs.

config key: svn-remote.<name>.ignore-refs

If the ignore-refs configuration key is set, and the command-line option is also given, both regular expressions will be used.

--ignore-paths=<regex>

This allows one to specify a Perl regular expression that will cause skipping of all matching paths from checkout from SVN. The *--ignore-paths* option should match for every *fetch* (including automatic fetches due to *clone*, *dcommit*, *rebase*, etc) on a given repository.

config key: svn-remote.<name>.ignore-paths

If the ignore-paths configuration key is set, and the command-line option is also given, both regular expressions will be used.

Examples:

Skip "doc*" directory for every fetch

```
--ignore-paths="^doc"
```

Skip "branches" and "tags" of first level directories

```
--ignore-paths="^[^/]+/(?:branches|tags)"
```

--include-paths=<regex>

This allows one to specify a Perl regular expression that will cause the inclusion of only matching paths from checkout from SVN. The *--include-paths* option should match for every *fetch* (including automatic fetches due to *clone*, *dcommit*, *rebase*, etc) on a given repository. *--ignore-paths* takes precedence over *--include-paths*.

config key: svn-remote.<name>.include-paths

--log-window-size=<n>

Fetch <n> log entries per request when scanning Subversion history. The default is 100. For very large Subversion repositories, larger values may be needed for *clone*/*fetch* to complete in reasonable time. But overly large values may lead to higher memory usage and request timeouts.

clone

Runs *init* and *fetch*. It will automatically create a directory based on the basename of the URL passed to it; or if a second argument is passed; it will create a directory and work within that. It accepts all arguments that the *init* and *fetch* commands accept; with the exception of *--fetch-all* and *--parent*. After a repository is cloned, the *fetch* command will be able to update revisions without affecting the working tree; and the *rebase* command will be able to update the working tree with the latest changes.

--preserve-empty-dirs

Create a placeholder file in the local Git repository for each empty directory fetched from Subversion. This includes directories that become empty by removing all entries in the Subversion repository (but not the directory itself). The placeholder files are also tracked and removed when no longer necessary.

--placeholder-filename=<filename>

Set the name of placeholder files created by --preserve-empty-dirs. Default: ".gitignore"

rebase

This fetches revisions from the SVN parent of the current HEAD and rebases the current (uncommitted to SVN) work against it.

This works similarly to *svn update* or *git pull* except that it preserves linear history with *git rebase* instead of *git merge* for ease of dcommitting with *git svn*.

This accepts all options that *git svn fetch* and *git rebase* accept. However, *--fetch-all* only fetches from the current [svn-remote], and not all [svn-remote] definitions.

Like *git rebase*; this requires that the working tree be clean and have no uncommitted changes.

This automatically updates the rev_map if needed (see *$GIT_DIR/svn/**/.rev_map.** in the FILES section below for details).

-l , --local

Do not fetch remotely; only run *git rebase* against the last fetched commit from the upstream SVN.

dcommit

Commit each diff from the current branch directly to the SVN repository, and then rebase or reset (depending on whether or not there is a diff between SVN and head). This will create a revision in SVN for each commit in Git.

When an optional Git branch name (or a Git commit object name) is specified as an argument, the subcommand works on the specified branch, not on the current branch.

Use of *dcommit* is preferred to *set-tree* (below).

--no-rebase

After committing, do not rebase or reset.

--commit-url <URL>

Commit to this SVN URL (the full path). This is intended to allow existing *git svn* repositories created with one transport method (e.g. *svn://* or *http://* for anonymous read) to be reused if a user is later given access to an alternate transport method (e.g. *svn+ssh://* or *https://*) for commit.

config key: svn-remote.<name>.commiturl
config key: svn.commiturl (overwrites all svn-remote.<name>.commiturl options)

Note that the SVN URL of the commiturl config key includes the SVN branch. If you rather want to set the commit URL for an entire SVN repository use svn-remote.<name>.pushurl instead.

Using this option for any other purpose (don't ask) is very strongly discouraged.

--mergeinfo=<mergeinfo>

Add the given merge information during the dcommit (e.g. *--mergeinfo="/branches/foo:1-10"*). All svn server versions can store this information (as a property), and svn clients starting from version 1.5 can make use of it. To specify merge information from multiple branches, use a single space character between the branches (*--mergeinfo="/branches/foo:1-10 /branches/bar:3,5-6,8"*)

config key: svn.pushmergeinfo

This option will cause git-svn to attempt to automatically populate the svn:mergeinfo property in the SVN repository when possible. Currently, this can only be done when dcommitting non-fast-forward merges where all parents but the first have already been pushed into SVN.

--interactive

Ask the user to confirm that a patch set should actually be sent to SVN. For each patch, one may answer "yes" (accept this patch), "no" (discard this patch), "all" (accept all patches), or "quit".

git svn dcommit returns immediately if answer is "no" or "quit", without committing anything to SVN.

branch

Create a branch in the SVN repository.

-m , --message

Allows to specify the commit message.

-t , --tag

Create a tag by using the tags_subdir instead of the branches_subdir specified during git svn init.

-d<path> , --destination=<path>

If more than one --branches (or --tags) option was given to the *init* or *clone* command, you must provide the location of the branch (or tag) you wish to create in the SVN repository. <path> specifies which path to use to create the branch or tag and should match the pattern on the left-hand side of one of the configured branches or tags refspecs. You can see these refspecs with the commands

```
git config --get-all svn-remote.<name>.branches
```

```
git config --get-all svn-remote.<name>.tags
```

where <name> is the name of the SVN repository as specified by the -R option to *init* (or "svn" by default).

--username

Specify the SVN username to perform the commit as. This option overrides the *username* configuration property.

--commit-url

Use the specified URL to connect to the destination Subversion repository. This is useful in cases where the source SVN repository is read-only. This option overrides configuration property *commiturl*.

```
git config --get-all svn-remote.<name>.commiturl
```

--parents

Create parent folders. This parameter is equivalent to the parameter --parents on svn cp commands and is useful for non-standard repository layouts.

tag

Create a tag in the SVN repository. This is a shorthand for *branch -t*.

log

This should make it easy to look up svn log messages when svn users refer to -r/--revision numbers.

The following features from svn log are supported:

-r <n>[:<n>] , --revision=<n>[:<n>]

is supported, non-numeric args are not: HEAD, NEXT, BASE, PREV, etc …

-v , --verbose

it's not completely compatible with the --verbose output in svn log, but reasonably close.

--limit=<n>

is NOT the same as --max-count, doesn't count merged/excluded commits

--incremental

supported

New features:

--show-commit

shows the Git commit sha1, as well

--oneline

our version of --pretty=oneline

> **Note**
>
> SVN itself only stores times in UTC and nothing else. The regular svn client converts the UTC time to the local time (or based on the TZ= environment). This command has the same behaviour.

Any other arguments are passed directly to *git log*

blame

Show what revision and author last modified each line of a file. The output of this mode is format-compatible with the output of svn blame by default. Like the SVN blame command, local uncommitted changes in the working tree are ignored; the version of the file in the HEAD revision is annotated. Unknown arguments are passed directly to *git blame*.

--git-format

Produce output in the same format as *git blame*, but with SVN revision numbers instead of Git commit hashes. In this mode, changes that haven't been committed to SVN (including local working-copy edits) are shown as revision 0.

find-rev

When given an SVN revision number of the form *rN*, returns the corresponding Git commit hash (this can optionally be followed by a tree-ish to specify which branch should be searched). When given a tree-ish, returns the corresponding SVN revision number.

-B , --before

Don't require an exact match if given an SVN revision, instead find the commit corresponding to the state of the SVN repository (on the current branch) at the specified revision.

-A , --after

Don't require an exact match if given an SVN revision; if there is not an exact match return the closest match searching forward in the history.

set-tree

You should consider using *dcommit* instead of this command. Commit specified commit or tree objects to SVN. This relies on your imported fetch data being up to date. This makes absolutely no attempts to do patching when committing to SVN, it simply overwrites files with those specified in the tree or commit. All merging is assumed to have taken place independently of *git svn* functions.

create-ignore

Recursively finds the svn:ignore property on directories and creates matching .gitignore files. The resulting files are staged to be committed, but are not committed. Use -r/--revision to refer to a specific revision.

show-ignore

Recursively finds and lists the svn:ignore property on directories. The output is suitable for appending to the $GIT_DIR/info/exclude file.

mkdirs

Attempts to recreate empty directories that core Git cannot track based on information in $GIT_DIR/svn/<refname>/unhandled.log files. Empty directories are automatically recreated when using "git svn clone" and "git svn rebase", so "mkdirs" is intended for use after commands like "git checkout" or "git reset". (See the svn-remote.<name>.automkdirs config file option for more information.)

commit-diff

Commits the diff of two tree-ish arguments from the command-line. This command does not rely on being inside a *git svn init*-ed repository. This command takes three arguments, (a) the original tree to diff against, (b) the new tree result, (c) the URL of the target Subversion repository. The final argument (URL) may be omitted if you are working from a *git svn*-aware repository (that has been *init*-ed with *git svn*). The -r<revision> option is required for this.

The commit message is supplied either directly with the *-m* or *-F* option, or indirectly from the tag or commit when the second tree-ish denotes such an object, or it is requested by invoking an editor (see *--edit* option below).

-m <msg> , --message=<msg>

Use the given *msg* as the commit message. This option disables the *--edit* option.

-F <filename> , --file=<filename>

Take the commit message from the given file. This option disables the *--edit* option.

info

Shows information about a file or directory similar to what svn info provides. Does not currently support a -r/--revision argument. Use the --url option to output only the value of the *URL:* field.

proplist

Lists the properties stored in the Subversion repository about a given file or directory. Use -r/--revision to refer to a specific Subversion revision.

propget

Gets the Subversion property given as the first argument, for a file. A specific revision can be specified with -r/--revision.

propset

Sets the Subversion property given as the first argument, to the value given as the second argument for the file given as the third argument.

Example:

```
git svn propset svn:keywords "FreeBSD=%H" devel/py-tipper/Makefile
```

This will set the property *svn:keywords* to *FreeBSD=%H* for the file *devel/py-tipper/Makefile*.

show-externals

Shows the Subversion externals. Use -r/--revision to specify a specific revision.

gc

Compress $GIT_DIR/svn/<refname>/unhandled.log files and remove $GIT_DIR/svn/<refname>/index files.

reset

Undoes the effects of *fetch* back to the specified revision. This allows you to re-*fetch* an SVN revision. Normally the contents of an SVN revision should never change and *reset* should not be necessary. However, if SVN permissions change, or if you alter your --ignore-paths option, a *fetch* may fail with "not found in commit" (file not previously visible) or "checksum mismatch" (missed a modification). If the problem file cannot be ignored forever (with --ignore-paths) the only way to repair the repo is to use *reset*.

Only the rev_map and refs/remotes/git-svn are changed (see *$GIT_DIR/svn/**/.rev_map.** in the FILES section below for details). Follow *reset* with a *fetch* and then *git reset* or *git rebase* to move local branches onto the new tree.

-r <n> , --revision=<n>

Specify the most recent revision to keep. All later revisions are discarded.

-p , --parent

Discard the specified revision as well, keeping the nearest parent instead.

Example:

Assume you have local changes in "master", but you need to refetch "r2".

```
r1---r2---r3 remotes/git-svn
             \
              A---B master
```

Fix the ignore-paths or SVN permissions problem that caused "r2" to be incomplete in the first place. Then:

```
git svn reset -r2 -p
git svn fetch
```

```
r1---r2'--r3' remotes/git-svn
  \
   r2---r3---A---B master
```

Then fixup "master" with *git rebase*. Do NOT use *git merge* or your history will not be compatible with a future *dcommit*!

```
git rebase --onto remotes/git-svn A^ master
```

```
r1---r2'--r3' remotes/git-svn
             \
              A'--B' master
```

OPTIONS

--shared[=(false|true|umask|group|all|world|everybody)] , --template=<template-directory>

Only used with the *init* command. These are passed directly to *git init*.

-r <arg> , --revision <arg>

Used with the *fetch* command.

This allows revision ranges for partial/cauterized history to be supported. $NUMBER, $NUMBER1:$NUMBER2 (numeric ranges), $NUMBER:HEAD, and BASE:$NUMBER are all supported.

This can allow you to make partial mirrors when running fetch; but is generally not recommended because history will be skipped and lost.

- , --stdin

Only used with the *set-tree* command.

Read a list of commits from stdin and commit them in reverse order. Only the leading sha1 is read from each line, so *git rev-list --pretty=oneline* output can be used.

--rmdir

Only used with the *dcommit*, *set-tree* and *commit-diff* commands.

Remove directories from the SVN tree if there are no files left behind. SVN can version empty directories, and they are not removed by default if there are no files left in them. Git cannot version empty directories. Enabling this flag will make the commit to SVN act like Git.

config key: svn.rmdir

-e , --edit

Only used with the *dcommit*, *set-tree* and *commit-diff* commands.

Edit the commit message before committing to SVN. This is off by default for objects that are commits, and forced on when committing tree objects.

config key: svn.edit

-l<num> , --find-copies-harder

Only used with the *dcommit*, *set-tree* and *commit-diff* commands.

They are both passed directly to *git diff-tree*; see Section G.3.43, “git-diff-tree(1)” for more information.

config key: svn.l
config key: svn.findcopiesharder

-A<filename> , --authors-file=<filename>

Syntax is compatible with the file used by *git cvsimport* but an empty email address can be supplied with <>:

```
loginname = Joe User <user@example.com>
```

If this option is specified and *git svn* encounters an SVN committer name that does not exist in the authors-file, *git svn* will abort operation. The user will then have to add the appropriate entry. Re-running the previous *git svn* command after the authors-file is modified should continue operation.

config key: svn.authorsfile

--authors-prog=<filename>

If this option is specified, for each SVN committer name that does not exist in the authors file, the given file is executed with the committer name as the first argument. The program is expected to return a single line of the form "Name <email>" or "Name <>", which will be treated as if included in the authors file.

Due to historical reasons a relative *filename* is first searched relative to the current directory for *init* and *clone* and relative to the root of the working tree for *fetch*. If *filename* is not found, it is searched like any other command in *$PATH*.

config key: svn.authorsProg

-q , --quiet

Make *git svn* less verbose. Specify a second time to make it even less verbose.

-m , --merge , -s<strategy> , --strategy=<strategy> , -p , --rebase-merges

These are only used with the *dcommit* and *rebase* commands.

Passed directly to *git rebase* when using *dcommit* if a *git reset* cannot be used (see *dcommit*).

-n , --dry-run

This can be used with the *dcommit*, *rebase*, *branch* and *tag* commands.

For *dcommit*, print out the series of Git arguments that would show which diffs would be committed to SVN.

For *rebase*, display the local branch associated with the upstream svn repository associated with the current branch and the URL of svn repository that will be fetched from.

For *branch* and *tag*, display the urls that will be used for copying when creating the branch or tag.

--use-log-author

When retrieving svn commits into Git (as part of *fetch*, *rebase*, or *dcommit* operations), look for the first *From:* line or *Signed-off-by* trailer in the log message and use that as the author string.

config key: svn.useLogAuthor

--add-author-from

When committing to svn from Git (as part of *set-tree* or *dcommit* operations), if the existing log message doesn't already have a *From:* or *Signed-off-by* trailer, append a *From:* line based on the Git commit's author string. If you use this, then *--use-log-author* will retrieve a valid author string for all commits.

config key: svn.addAuthorFrom

ADVANCED OPTIONS

-i<GIT_SVN_ID> , --id <GIT_SVN_ID>

This sets GIT_SVN_ID (instead of using the environment). This allows the user to override the default refname to fetch from when tracking a single URL. The *log* and *dcommit* commands no longer require this switch as an argument.

-R<remote-name> , --svn-remote <remote-name>

Specify the [svn-remote "<remote-name>"] section to use, this allows SVN multiple repositories to be tracked. Default: "svn"

--follow-parent

This option is only relevant if we are tracking branches (using one of the repository layout options --trunk, --tags, --branches, --stdlayout). For each tracked branch, try to find out where its revision was copied from, and set a suitable parent in the first Git commit for the branch. This is especially helpful when we're tracking a directory that has been moved around within the repository. If this feature is disabled, the branches created by *git svn* will all be linear and not share any history, meaning that there will be no information on where branches were branched off or merged. However, following long/convoluted histories can take a long time, so disabling this feature may speed up the cloning process. This feature is enabled by default, use --no-follow-parent to disable it.

config key: svn.followparent

CONFIG FILE-ONLY OPTIONS

svn.noMetadata , svn-remote.<name>.noMetadata

This gets rid of the *git-svn-id:* lines at the end of every commit.

This option can only be used for one-shot imports as *git svn* will not be able to fetch again without metadata. Additionally, if you lose your *$GIT_DIR/svn/**/.rev_map.** files, *git svn* will not be able to rebuild them.

The *git svn log* command will not work on repositories using this, either. Using this conflicts with the *useSvmProps* option for (hopefully) obvious reasons.

This option is NOT recommended as it makes it difficult to track down old references to SVN revision numbers in existing documentation, bug reports, and archives. If you plan to eventually migrate from SVN to Git and are certain about dropping SVN history, consider git-filter-repo [https://github.com/newren/git-filter-repo] instead. filter-repo also allows reformatting of metadata for ease-of-reading and rewriting authorship info for non-"svn.authorsFile" users.

svn.useSvmProps , svn-remote.<name>.useSvmProps

This allows *git svn* to re-map repository URLs and UUIDs from mirrors created using SVN::Mirror (or svk) for metadata.

If an SVN revision has a property, "svm:headrev", it is likely that the revision was created by SVN::Mirror (also used by SVK). The property contains a repository UUID and a revision. We want to make it look like we are mirroring the original URL, so introduce a helper function that returns the original identity URL and UUID, and use it when generating metadata in commit messages.

svn.useSvnsyncProps , svn-remote.<name>.useSvnsyncprops

Similar to the useSvmProps option; this is for users of the svnsync(1) command distributed with SVN 1.4.x and later.

svn-remote.<name>.rewriteRoot

This allows users to create repositories from alternate URLs. For example, an administrator could run *git svn* on the server locally (accessing via file://) but wish to distribute the repository with a public http:// or svn:// URL in the metadata so users of it will see the public URL.

svn-remote.<name>.rewriteUUID

Similar to the useSvmProps option; this is for users who need to remap the UUID manually. This may be useful in situations where the original UUID is not available via either useSvmProps or useSvnsyncProps.

svn-remote.<name>.pushurl

Similar to Git's *remote.<name>.pushurl*, this key is designed to be used in cases where *url* points to an SVN repository via a read-only transport, to provide an alternate read/write transport. It is assumed that both keys point to the same repository. Unlike *commiturl*, *pushurl* is a base path. If either *commiturl* or *pushurl* could be used, *commiturl* takes precedence.

svn.brokenSymlinkWorkaround

This disables potentially expensive checks to workaround broken symlinks checked into SVN by broken clients. Set this option to "false" if you track a SVN repository with many empty blobs that are not symlinks. This option may be changed while *git svn* is running and take effect on the next revision fetched. If unset, *git svn* assumes this option to be "true".

svn.pathnameencoding

This instructs git svn to recode pathnames to a given encoding. It can be used by windows users and by those who work in non-utf8 locales to avoid corrupted file names with non-ASCII characters. Valid encodings are the ones supported by Perl's Encode module.

svn-remote.<name>.automkdirs

Normally, the "git svn clone" and "git svn rebase" commands attempt to recreate empty directories that are in the Subversion repository. If this option is set to "false", then empty directories will only be created if the "git svn mkdirs" command is run explicitly. If unset, *git svn* assumes this option to be "true".

Since the noMetadata, rewriteRoot, rewriteUUID, useSvnsyncProps and useSvmProps options all affect the metadata generated and used by *git svn*; they **must** be set in the configuration file before any history is imported and these settings should never be changed once they are set.

Additionally, only one of these options can be used per svn-remote section because they affect the *git-svn-id:* metadata line, except for rewriteRoot and rewriteUUID which can be used together.

BASIC EXAMPLES

Tracking and contributing to the trunk of a Subversion-managed project (ignoring tags and branches):

```
# Clone a repo (like git clone):
        git svn clone http://svn.example.com/project/trunk
# Enter the newly cloned directory:
        cd trunk
# You should be on master branch, double-check with 'git branch'
        git branch
# Do some work and commit locally to Git:
        git commit ...
# Something is committed to SVN, rebase your local changes against the
# latest changes in SVN:
        git svn rebase
# Now commit your changes (that were committed previously using Git) to
 SVN,
# as well as automatically updating your working HEAD:
        git svn dcommit
# Append svn:ignore settings to the default Git exclude file:
        git svn show-ignore >> .git/info/exclude
```

Tracking and contributing to an entire Subversion-managed project (complete with a trunk, tags and branches):

```
# Clone a repo with standard SVN directory layout (like git clone):
        git svn clone http://svn.example.com/project --stdlayout --prefix
 svn/
# Or, if the repo uses a non-standard directory layout:
        git svn clone http://svn.example.com/project -T tr -b branch -t tag
 --prefix svn/
# View all branches and tags you have cloned:
        git branch -r
# Create a new branch in SVN
        git svn branch waldo
# Reset your master to trunk (or any other branch, replacing 'trunk'
# with the appropriate name):
        git reset --hard svn/trunk
# You may only dcommit to one branch/tag/trunk at a time.  The usage
# of dcommit/rebase/show-ignore should be the same as above.
```

The initial *git svn clone* can be quite time-consuming (especially for large Subversion repositories). If multiple people (or one person with multiple machines) want to use *git svn* to interact with the same Subversion repository, you can do the initial *git svn clone* to a repository on a server and have each person clone that repository with *git clone*:

```
# Do the initial import on a server
        ssh server "cd /pub && git svn clone http://svn.example.com/project
 [options...]"
# Clone locally - make sure the refs/remotes/ space matches the server
        mkdir project
        cd project
        git init
        git remote add origin server:/pub/project
        git config --replace-all remote.origin.fetch '+refs/remotes/*:refs/
remotes/*'
        git fetch
# Prevent fetch/pull from remote Git server in the future,
# we only want to use git svn for future updates
        git config --remove-section remote.origin
# Create a local branch from one of the branches just fetched
        git checkout -b master FETCH_HEAD
# Initialize 'git svn' locally (be sure to use the same URL and
# --stdlayout/-T/-b/-t/--prefix options as were used on server)
```

```
        git svn init http://svn.example.com/project [options...]
# Pull the latest changes from Subversion
        git svn rebase
```

REBASE VS. PULL/MERGE

Prefer to use *git svn rebase* or *git rebase*, rather than *git pull* or *git merge* to synchronize unintegrated commits with a *git svn* branch. Doing so will keep the history of unintegrated commits linear with respect to the upstream SVN repository and allow the use of the preferred *git svn dcommit* subcommand to push unintegrated commits back into SVN.

Originally, *git svn* recommended that developers pulled or merged from the *git svn* branch. This was because the author favored *git svn set-tree B* to commit a single head rather than the *git svn set-tree A..B* notation to commit multiple commits. Use of *git pull* or *git merge* with *git svn set-tree A..B* will cause non-linear history to be flattened when committing into SVN and this can lead to merge commits unexpectedly reversing previous commits in SVN.

MERGE TRACKING

While *git svn* can track copy history (including branches and tags) for repositories adopting a standard layout, it cannot yet represent merge history that happened inside git back upstream to SVN users. Therefore it is advised that users keep history as linear as possible inside Git to ease compatibility with SVN (see the CAVEATS section below).

HANDLING OF SVN BRANCHES

If *git svn* is configured to fetch branches (and --follow-branches is in effect), it sometimes creates multiple Git branches for one SVN branch, where the additional branches have names of the form *branchname@nnn* (with nnn an SVN revision number). These additional branches are created if *git svn* cannot find a parent commit for the first commit in an SVN branch, to connect the branch to the history of the other branches.

Normally, the first commit in an SVN branch consists of a copy operation. *git svn* will read this commit to get the SVN revision the branch was created from. It will then try to find the Git commit that corresponds to this SVN revision, and use that as the parent of the branch. However, it is possible that there is no suitable Git commit to serve as parent. This will happen, among other reasons, if the SVN branch is a copy of a revision that was not fetched by *git svn* (e.g. because it is an old revision that was skipped with *--revision*), or if in SVN a directory was copied that is not tracked by *git svn* (such as a branch that is not tracked at all, or a subdirectory of a tracked branch). In these cases, *git svn* will still create a Git branch, but instead of using an existing Git commit as the parent of the branch, it will read the SVN history of the directory the branch was copied from and create appropriate Git commits. This is indicated by the message "Initializing parent: <branchname>".

Additionally, it will create a special branch named *<branchname>@<SVN-Revision>*, where <SVN-Revision> is the SVN revision number the branch was copied from. This branch will point to the newly created parent commit of the branch. If in SVN the branch was deleted and later recreated from a different version, there will be multiple such branches with an @.

Note that this may mean that multiple Git commits are created for a single SVN revision.

An example: in an SVN repository with a standard trunk/tags/branches layout, a directory trunk/sub is created in r.100. In r.200, trunk/sub is branched by copying it to branches/. *git svn clone -s* will then create a branch *sub*. It will also create new Git commits for r.100 through r.199 and use these as the history of branch *sub*. Thus there will be two Git commits for each revision from r.100 to r.199 (one containing trunk/, one containing trunk/sub/). Finally, it will create a branch *sub@200* pointing to the new parent commit of branch *sub* (i.e. the commit for r.200 and trunk/sub/).

CAVEATS

For the sake of simplicity and interoperating with Subversion, it is recommended that all *git svn* users clone, fetch and dcommit directly from the SVN server, and avoid all *git clone*/*pull*/*merge*/*push* operations between Git repositories and branches. The recommended method of exchanging code between Git branches and users is *git format-patch* and *git am*, or just 'dcommit'ing to the SVN repository.

Running *git merge* or *git pull* is NOT recommended on a branch you plan to *dcommit* from because Subversion users cannot see any merges you've made. Furthermore, if you merge or pull from a Git branch that is a mirror of an SVN branch, *dcommit* may commit to the wrong branch.

If you do merge, note the following rule: *git svn dcommit* will attempt to commit on top of the SVN commit named in

```
git log --grep=^git-svn-id: --first-parent -1
```

You *must* therefore ensure that the most recent commit of the branch you want to dcommit to is the *first* parent of the merge. Chaos will ensue otherwise, especially if the first parent is an older commit on the same SVN branch.

git clone does not clone branches under the refs/remotes/ hierarchy or any *git svn* metadata, or config. So repositories created and managed with using *git svn* should use *rsync* for cloning, if cloning is to be done at all.

Since *dcommit* uses rebase internally, any Git branches you *git push* to before *dcommit* on will require forcing an overwrite of the existing ref on the remote repository. This is generally considered bad practice, see the Section G.3.103, “git-push(1)” documentation for details.

Do not use the --amend option of Section G.3.28, “git-commit(1)” on a change you've already dcommitted. It is considered bad practice to --amend commits you've already pushed to a remote repository for other users, and dcommit with SVN is analogous to that.

When cloning an SVN repository, if none of the options for describing the repository layout is used (--trunk, --tags, --branches, --stdlayout), *git svn clone* will create a Git repository with completely linear history, where branches and tags appear as separate directories in the working copy. While this is the easiest way to get a copy of a complete repository, for projects with many branches it will lead to a working copy many times larger than just the trunk. Thus for projects using the standard directory structure (trunk/branches/tags), it is recommended to clone with option *--stdlayout*. If the project uses a non-standard structure, and/or if branches and tags are not required, it is easiest to only clone one directory (typically trunk), without giving any repository layout options. If the full history with branches and tags is required, the options *--trunk* / *--branches* / *--tags* must be used.

When using multiple --branches or --tags, *git svn* does not automatically handle name collisions (for example, if two branches from different paths have the same name, or if a branch and a tag have the same name). In these cases, use *init* to set up your Git repository then, before your first *fetch*, edit the $GIT_DIR/config file so that the branches and tags are associated with different name spaces. For example:

```
branches = stable/*:refs/remotes/svn/stable/*
branches = debug/*:refs/remotes/svn/debug/*
```

CONFIGURATION

git svn stores [svn-remote] configuration information in the repository $GIT_DIR/config file. It is similar the core Git [remote] sections except *fetch* keys do not accept glob arguments; but they are instead handled by the *branches* and *tags* keys. Since some SVN repositories are oddly configured with multiple projects glob expansions such those listed below are allowed:

```
[svn-remote "project-a"]
        url = http://server.org/svn
        fetch = trunk/project-a:refs/remotes/project-a/trunk
        branches = branches/*/project-a:refs/remotes/project-a/branches/*
        branches = branches/release_*:refs/remotes/project-a/branches/
release_*
        branches = branches/re*se:refs/remotes/project-a/branches/*
        tags = tags/*/project-a:refs/remotes/project-a/tags/*
```

Keep in mind that the * (asterisk) wildcard of the local ref (right of the *:*) **must** be the farthest right path component; however the remote wildcard may be anywhere as long as it's an independent path component (surrounded by / or EOL). This type of configuration is not automatically created by *init* and should be manually entered with a text-editor or using *git config*.

Also note that only one asterisk is allowed per word. For example:

```
branches = branches/re*se:refs/remotes/project-a/branches/*
```

will match branches *release*, *rese*, *re123se*, however

```
branches = branches/re*s*e:refs/remotes/project-a/branches/*
```

will produce an error.

It is also possible to fetch a subset of branches or tags by using a comma-separated list of names within braces. For example:

```
[svn-remote "huge-project"]
        url = http://server.org/svn
        fetch = trunk/src:refs/remotes/trunk
        branches = branches/{red,green}/src:refs/remotes/project-a/
branches/*
        tags = tags/{1.0,2.0}/src:refs/remotes/project-a/tags/*
```

Multiple fetch, branches, and tags keys are supported:

```
[svn-remote "messy-repo"]
        url = http://server.org/svn
        fetch = trunk/project-a:refs/remotes/project-a/trunk
        fetch = branches/demos/june-project-a-demo:refs/remotes/project-a/
demos/june-demo
        branches = branches/server/*:refs/remotes/project-a/branches/*
        branches = branches/demos/2011/*:refs/remotes/project-a/2011-demos/
*
        tags = tags/server/*:refs/remotes/project-a/tags/*
```

Creating a branch in such a configuration requires disambiguating which location to use using the -d or --destination flag:

```
$ git svn branch -d branches/server release-2-3-0
```

Note that git-svn keeps track of the highest revision in which a branch or tag has appeared. If the subset of branches or tags is changed after fetching, then $GIT_DIR/svn/.metadata must be manually edited to remove (or reset) branches-maxRev and/or tags-maxRev as appropriate.

FILES

$GIT_DIR/svn/**/.rev_map.*

> Mapping between Subversion revision numbers and Git commit names. In a repository where the noMetadata option is not set, this can be rebuilt from the git-svn-id: lines that are at the end of every commit (see the *svn.noMetadata* section above for details).
>
> *git svn fetch* and *git svn rebase* automatically update the rev_map if it is missing or not up to date. *git svn reset* automatically rewinds it.

BUGS

We ignore all SVN properties except svn:executable. Any unhandled properties are logged to $GIT_DIR/svn/<refname>/unhandled.log

Renamed and copied directories are not detected by Git and hence not tracked when committing to SVN. I do not plan on adding support for this as it's quite difficult and time-consuming to get working for all the possible corner cases (Git doesn't do it, either). Committing renamed and copied files is fully supported if they're similar enough for Git to detect them.

In SVN, it is possible (though discouraged) to commit changes to a tag (because a tag is just a directory copy, thus technically the same as a branch). When cloning an SVN repository, *git svn* cannot know if such a commit to a tag will happen in the future. Thus it acts conservatively and imports all SVN tags as branches, prefixing the tag name with *tags/*.

SEE ALSO

Section G.3.107, "git-rebase(1)"

GIT

Part of the Section G.3.1, "git(1)" suite

G.3.143. git-symbolic-ref(1)

2024-07-02
Revision History
2024-07-02

NAME

git-symbolic-ref - Read, modify and delete symbolic refs

SYNOPSIS

git symbolic-ref [-m <reason>] <name> <ref>
git symbolic-ref [-q] [--short] [--no-recurse] <name>
git symbolic-ref --delete [-q] <name>

DESCRIPTION

Given one argument, reads which branch head the given symbolic ref refers to and outputs its path, relative to the *.git/* directory. Typically you would give *HEAD* as the <name> argument to see which branch your working tree is on.

Given two arguments, creates or updates a symbolic ref <name> to point at the given branch <ref>.

Given *--delete* and an additional argument, deletes the given symbolic ref.

A symbolic ref is a regular file that stores a string that begins with *ref: refs/*. For example, your *.git/HEAD* is a regular file whose content is *ref: refs/heads/master*.

OPTIONS

-d , --delete

Delete the symbolic ref <name>.

-q , --quiet

Do not issue an error message if the <name> is not a symbolic ref but a detached HEAD; instead exit with non-zero status silently.

--short

When showing the value of <name> as a symbolic ref, try to shorten the value, e.g. from *refs/heads/master* to *master*.

--recurse , --no-recurse

When showing the value of <name> as a symbolic ref, if <name> refers to another symbolic ref, follow such a chain of symbolic refs until the result no longer points at a symbolic ref (*--recurse*, which is the default). *--no-recurse* stops after dereferencing only a single level of symbolic ref.

-m

Update the reflog for <name> with <reason>. This is valid only when creating or updating a symbolic ref.

NOTES

In the past, *.git/HEAD* was a symbolic link pointing at *refs/heads/master*. When we wanted to switch to another branch, we did *ln -sf refs/heads/newbranch .git/HEAD*, and when we wanted to find out which branch we are on, we did *readlink .git/HEAD*. But symbolic links are not entirely portable, so they are now deprecated and symbolic refs (as described above) are used by default.

git symbolic-ref will exit with status 0 if the contents of the symbolic ref were printed correctly, with status 1 if the requested name is not a symbolic ref, or 128 if another error occurs.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.144. git-tag(1)

2024-07-02
Revision History
2024-07-02

NAME

git-tag - Create, list, delete or verify a tag object signed with GPG

SYNOPSIS

git tag [-a | -s | -u <key-id>] [-f] [-m <msg> | -F <file>] [-e]
<tagname> [<commit> | <object>]
git tag -d <tagname>…
git tag [-n[<num>]] -l [--contains <commit>] [--no-contains <commit>]
[--points-at <object>] [--column[=<options>] | --no-column]
[--create-reflog] [--sort=<key>] [--format=<format>]
[--merged <commit>] [--no-merged <commit>] [<pattern>…]
git tag -v [--format=<format>] <tagname>…

DESCRIPTION

Add a tag reference in *refs/tags/*, unless *-d/-l/-v* is given to delete, list or verify tags.

Unless *-f* is given, the named tag must not yet exist.

If one of *-a*, *-s*, or *-u <key-id>* is passed, the command creates a *tag* object, and requires a tag message. Unless *-m <msg>* or *-F <file>* is given, an editor is started for the user to type in the tag message.

If *-m <msg>* or *-F <file>* is given and *-a*, *-s*, and *-u <key-id>* are absent, *-a* is implied.

Otherwise, a tag reference that points directly at the given object (i.e., a lightweight tag) is created.

A GnuPG signed tag object will be created when *-s* or *-u <key-id>* is used. When *-u <key-id>* is not used, the committer identity for the current user is used to find the GnuPG key for signing. The configuration variable *gpg.program* is used to specify custom GnuPG binary.

Tag objects (created with *-a*, *-s*, or *-u*) are called "annotated" tags; they contain a creation date, the tagger name and e-mail, a tagging message, and an optional GnuPG signature. Whereas a "lightweight" tag is simply a name for an object (usually a commit object).

Annotated tags are meant for release while lightweight tags are meant for private or temporary object labels. For this reason, some git commands for naming objects (like *git describe*) will ignore lightweight tags by default.

OPTIONS

-a , --annotate

Make an unsigned, annotated tag object

-s , --sign

Make a GPG-signed tag, using the default e-mail address's key. The default behavior of tag GPG-signing is controlled by *tag.gpgSign* configuration variable if it exists, or disabled otherwise. See Section G.3.29, “git-config(1)”.

--no-sign

Override *tag.gpgSign* configuration variable that is set to force each and every tag to be signed.

-u <key-id> , --local-user=<key-id>

Make a GPG-signed tag, using the given key.

-f , --force

Replace an existing tag with the given name (instead of failing)

-d , --delete

Delete existing tags with the given names.

-v , --verify

Verify the GPG signature of the given tag names.

-n<num>

<num> specifies how many lines from the annotation, if any, are printed when using -l. Implies *--list*.

The default is not to print any annotation lines. If no number is given to *-n*, only the first line is printed. If the tag is not annotated, the commit message is displayed instead.

-l , --list

List tags. With optional *<pattern>...*, e.g. *git tag --list 'v-*'*, list only the tags that match the pattern(s).

Running "git tag" without arguments also lists all tags. The pattern is a shell wildcard (i.e., matched using fnmatch(3)). Multiple patterns may be given; if any of them matches, the tag is shown.

This option is implicitly supplied if any other list-like option such as *--contains* is provided. See the documentation for each of those options for details.

--sort=<key>

Sort based on the key given. Prefix - to sort in descending order of the value. You may use the --sort=<key> option multiple times, in which case the last key becomes the primary key. Also supports "version:refname" or "v:refname" (tag names are treated as versions). The "version:refname" sort order can also be affected by the "versionsort.suffix" configuration variable. The keys supported are the same as those in *git for-each-ref*. Sort order defaults to the value configured for the *tag.sort* variable if it exists, or lexicographic order otherwise. See Section G.3.29, “git-config(1)”.

--color[=<when>]

Respect any colors specified in the *--format* option. The *<when>* field must be one of *always*, *never*, or *auto* (if *<when>* is absent, behave as if *always* was given).

-i , --ignore-case

Sorting and filtering tags are case insensitive.

--omit-empty

Do not print a newline after formatted refs where the format expands to the empty string.

--column[=<options>] , --no-column

Display tag listing in columns. See configuration variable *column.tag* for option syntax. *--column* and *--no-column* without options are equivalent to *always* and *never* respectively.

This option is only applicable when listing tags without annotation lines.

--contains [<commit>]

Only list tags which contain the specified commit (HEAD if not specified). Implies *--list*.

--no-contains [<commit>]

Only list tags which don't contain the specified commit (HEAD if not specified). Implies *--list*.

--merged [<commit>]

Only list tags whose commits are reachable from the specified commit (*HEAD* if not specified).

--no-merged [<commit>]

Only list tags whose commits are not reachable from the specified commit (*HEAD* if not specified).

--points-at <object>

Only list tags of the given object (HEAD if not specified). Implies *--list*.

-m <msg> , --message=<msg>

Use the given tag message (instead of prompting). If multiple *-m* options are given, their values are concatenated as separate paragraphs. Implies *-a* if none of *-a*, *-s*, or *-u* <*key-id*> is given.

-F <file> , --file=<file>

Take the tag message from the given file. Use - to read the message from the standard input. Implies *-a* if none of *-a*, *-s*, or *-u* <*key-id*> is given.

-e , --edit

The message taken from file with *-F* and command line with *-m* are usually used as the tag message unmodified. This option lets you further edit the message taken from these sources.

--cleanup=<mode>

This option sets how the tag message is cleaned up. The <*mode*> can be one of *verbatim*, *whitespace* and *strip*. The *strip* mode is default. The *verbatim* mode does not change message at all, *whitespace* removes just leading/trailing whitespace lines and *strip* removes both whitespace and commentary.

--create-reflog

Create a reflog for the tag. To globally enable reflogs for tags, see *core.logAllRefUpdates* in Section G.3.29, “git-config(1)”. The negated form *--no-create-reflog* only overrides an earlier *--create-reflog*, but currently does not negate the setting of *core.logAllRefUpdates*.

--format=<format>

A string that interpolates *%(fieldname)* from a tag ref being shown and the object it points at. The format is the same as that of Section G.3.52, “git-for-each-ref(1)”. When unspecified, defaults to *%(refname:strip=2)*.

<tagname>

The name of the tag to create, delete, or describe. The new tag name must pass all checks defined by Section G.3.17, “git-check-ref-format(1)”. Some of these checks may restrict the characters allowed in a tag name.

<commit> , <object>

The object that the new tag will refer to, usually a commit. Defaults to HEAD.

CONFIGURATION

By default, *git tag* in sign-with-default mode (-s) will use your committer identity (of the form *Your Name <your@email.address>*) to find a key. If you want to use a different default key, you can specify it in the repository configuration as follows:

```
[user]
    signingKey = <gpg-key-id>
```

pager.tag is only respected when listing tags, i.e., when *-l* is used or implied. The default is to use a pager. See Section G.3.29, “git-config(1)”.

DISCUSSION

1. On Re-tagging

What should you do when you tag a wrong commit and you would want to re-tag?

If you never pushed anything out, just re-tag it. Use "-f" to replace the old one. And you're done.

But if you have pushed things out (or others could just read your repository directly), then others will have already seen the old tag. In that case you can do one of two things:

1. The sane thing. Just admit you screwed up, and use a different name. Others have already seen one tag-name, and if you keep the same name, you may be in the situation that two people both have "version X", but they actually have *different* "X"'s. So just call it "X.1" and be done with it.

2. The insane thing. You really want to call the new version "X" too, *even though* others have already seen the old one. So just use *git tag -f* again, as if you hadn't already published the old one.

However, Git does **not** (and it should not) change tags behind users back. So if somebody already got the old tag, doing a *git pull* on your tree shouldn't just make them overwrite the old one.

If somebody got a release tag from you, you cannot just change the tag for them by updating your own one. This is a big security issue, in that people MUST be able to trust their tag-names. If you really want to do the insane thing, you need to just fess up to it, and tell people that you messed up. You can do that by making a very public announcement saying:

```
Ok, I messed up, and I pushed out an earlier version tagged as X. I
then fixed something, and retagged the *fixed* tree as X again.

If you got the wrong tag, and want the new one, please delete
the old one and fetch the new one by doing:

        git tag -d X
        git fetch origin tag X
```

```
to get my updated tag.

You can test which tag you have by doing

        git rev-parse X

which should return 0123456789abcdef.. if you have the new version.

Sorry for the inconvenience.
```

Does this seem a bit complicated? It **should** be. There is no way that it would be correct to just "fix" it automatically. People need to know that their tags might have been changed.

2. On Automatic following

If you are following somebody else's tree, you are most likely using remote-tracking branches (eg. *refs/remotes/origin/master*). You usually want the tags from the other end.

On the other hand, if you are fetching because you would want a one-shot merge from somebody else, you typically do not want to get tags from there. This happens more often for people near the toplevel but not limited to them. Mere mortals when pulling from each other do not necessarily want to automatically get private anchor point tags from the other person.

Often, "please pull" messages on the mailing list just provide two pieces of information: a repo URL and a branch name; this is designed to be easily cut&pasted at the end of a *git fetch* command line:

```
Linus, please pull from

        git://git..../proj.git master

to get the following updates...
```

becomes:

```
$ git pull git://git..../proj.git master
```

In such a case, you do not want to automatically follow the other person's tags.

One important aspect of Git is its distributed nature, which largely means there is no inherent "upstream" or "downstream" in the system. On the face of it, the above example might seem to indicate that the tag namespace is owned by the upper echelon of people and that tags only flow downwards, but that is not the case. It only shows that the usage pattern determines who are interested in whose tags.

A one-shot pull is a sign that a commit history is now crossing the boundary between one circle of people (e.g. "people who are primarily interested in the networking part of the kernel") who may have their own set of tags (e.g. "this is the third release candidate from the networking group to be proposed for general consumption with 2.6.21 release") to another circle of people (e.g. "people who integrate various subsystem improvements"). The latter are usually not interested in the detailed tags used internally in the former group (that is what "internal" means). That is why it is desirable not to follow tags automatically in this case.

It may well be that among networking people, they may want to exchange the tags internal to their group, but in that workflow they are most likely tracking each other's progress by having remote-tracking branches. Again, the heuristic to automatically follow such tags is a good thing.

3. On Backdating Tags

If you have imported some changes from another VCS and would like to add tags for major releases of your work, it is useful to be able to specify the date to embed inside of the tag object; such data in the tag object affects, for example, the ordering of tags in the gitweb interface.

To set the date used in future tag objects, set the environment variable GIT_COMMITTER_DATE (see the later discussion of possible values; the most common form is "YYYY-MM-DD HH:MM").

For example:

```
$ GIT_COMMITTER_DATE="2006-10-02 10:31" git tag -s v1.0.1
```

DATE FORMATS

The *GIT_AUTHOR_DATE* and *GIT_COMMITTER_DATE* environment variables support the following date formats:

Git internal format

It is <*unix-timestamp*> <*time-zone-offset*>, where <*unix-timestamp*> is the number of seconds since the UNIX epoch. <*time-zone-offset*> is a positive or negative offset from UTC. For example CET (which is 1 hour ahead of UTC) is *+0100*.

RFC 2822

The standard date format as described by RFC 2822, for example *Thu, 07 Apr 2005 22:13:13 +0200*.

ISO 8601

Time and date specified by the ISO 8601 standard, for example *2005-04-07T22:13:13*. The parser accepts a space instead of the *T* character as well. Fractional parts of a second will be ignored, for example *2005-04-07T22:13:13.019* will be treated as *2005-04-07T22:13:13*.

> **Note**
>
> In addition, the date part is accepted in the following formats: *YYYY.MM.DD*, *MM/DD/YYYY* and *DD.MM.YYYY*.

FILES

$GIT_DIR/TAG_EDITMSG

This file contains the message of an in-progress annotated tag. If *git tag* exits due to an error before creating an annotated tag then the tag message that has been provided by the user in an editor session will be available in this file, but may be overwritten by the next invocation of *git tag*.

NOTES

When combining multiple *--contains* and *--no-contains* filters, only references that contain at least one of the *--contains* commits and contain none of the *--no-contains* commits are shown.

When combining multiple *--merged* and *--no-merged* filters, only references that are reachable from at least one of the *--merged* commits and from none of the *--no-merged* commits are shown.

SEE ALSO

Section G.3.17, “git-check-ref-format(1)”. Section G.3.29, “git-config(1)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.145. git-unpack-file(1)

2024-07-02

Revision History

2024-07-02

NAME

git-unpack-file - Creates a temporary file with a blob's contents

SYNOPSIS

git unpack-file <blob>

DESCRIPTION

Creates a file holding the contents of the blob specified by sha1. It returns the name of the temporary file in the following format: .merge_file_XXXXX

OPTIONS

<blob>

Must be a blob id

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.146. git-unpack-objects(1)

2024-07-02
Revision History
2024-07-02

NAME

git-unpack-objects - Unpack objects from a packed archive

SYNOPSIS

git unpack-objects [-n] [-q] [-r] [--strict]

DESCRIPTION

Read a packed archive (.pack) from the standard input, expanding the objects contained within and writing them into the repository in "loose" (one object per file) format.

Objects that already exist in the repository will **not** be unpacked from the packfile. Therefore, nothing will be unpacked if you use this command on a packfile that exists within the target repository.

See Section G.3.113, “git-repack(1)” for options to generate new packs and replace existing ones.

OPTIONS

-n

Dry run. Check the pack file without actually unpacking the objects.

-q

The command usually shows percentage progress. This flag suppresses it.

-r

When unpacking a corrupt packfile, the command dies at the first corruption. This flag tells it to keep going and make the best effort to recover as many objects as possible.

--strict

Don't write objects with broken content or links.

--max-input-size=<size>

Die, if the pack is larger than <size>.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.147. git-update-index(1)

2024-07-02
Revision History
2024-07-02

NAME

git-update-index - Register file contents in the working tree to the index

SYNOPSIS

git update-index
[--add] [--remove | --force-remove] [--replace]
[--refresh] [-q] [--unmerged] [--ignore-missing]
[(--cacheinfo <mode>,<object>,<file>)…]
[--chmod=(+|-)x]
[--[no-]assume-unchanged]
[--[no-]skip-worktree]
[--[no-]ignore-skip-worktree-entries]
[--[no-]fsmonitor-valid]
[--ignore-submodules]
[--[no-]split-index]
[--[no-|test-|force-]untracked-cache]
[--[no-]fsmonitor]
[--really-refresh] [--unresolve] [--again | -g]
[--info-only] [--index-info]
[-z] [--stdin] [--index-version <n>]
[--show-index-version]
[--verbose]
[--] [<file>…]

DESCRIPTION

Modifies the index. Each file mentioned is updated into the index and any *unmerged* or *needs updating* state is cleared.

See also Section G.3.2, “git-add(1)” for a more user-friendly way to do some of the most common operations on the index.

The way *git update-index* handles files it is told about can be modified using the various options:

OPTIONS

--add

If a specified file isn't in the index already then it's added. Default behaviour is to ignore new files.

--remove

If a specified file is in the index but is missing then it's removed. Default behavior is to ignore removed files.

--refresh

Looks at the current index and checks to see if merges or updates are needed by checking stat() information.

-q

Quiet. If --refresh finds that the index needs an update, the default behavior is to error out. This option makes *git update-index* continue anyway.

--ignore-submodules

Do not try to update submodules. This option is only respected when passed before --refresh.

--unmerged

If --refresh finds unmerged changes in the index, the default behavior is to error out. This option makes *git update-index* continue anyway.

--ignore-missing

Ignores missing files during a --refresh

--cacheinfo <mode>,<object>,<path> , --cacheinfo <mode> <object> <path>

Directly insert the specified info into the index. For backward compatibility, you can also give these three arguments as three separate parameters, but new users are encouraged to use a single-parameter form.

--index-info

Read index information from stdin.

--chmod=(+|-)x

Set the execute permissions on the updated files.

--[no-]assume-unchanged

When this flag is specified, the object names recorded for the paths are not updated. Instead, this option sets/unsets the "assume unchanged" bit for the paths. When the "assume unchanged" bit is on, the user promises not to change the file and allows Git to assume that the working tree file matches what is recorded in the index. If you want to change the working tree file, you need to unset the bit to tell Git. This is sometimes helpful when working with a big project on a filesystem that has a very slow lstat(2) system call (e.g. cifs).

Git will fail (gracefully) in case it needs to modify this file in the index e.g. when merging in a commit; thus, in case the assumed-untracked file is changed upstream, you will need to handle the situation manually.

--really-refresh

Like *--refresh*, but checks stat information unconditionally, without regard to the "assume unchanged" setting.

--[no-]skip-worktree

When one of these flags is specified, the object names recorded for the paths are not updated. Instead, these options set and unset the "skip-worktree" bit for the paths. See section "Skip-worktree bit" below for more information.

--[no-]ignore-skip-worktree-entries

Do not remove skip-worktree (AKA "index-only") entries even when the *--remove* option was specified.

--[no-]fsmonitor-valid

When one of these flags is specified, the object names recorded for the paths are not updated. Instead, these options set and unset the "fsmonitor valid" bit for the paths. See section "File System Monitor" below for more information.

-g , --again

Runs *git update-index* itself on the paths whose index entries are different from those of the *HEAD* commit.

--unresolve

Restores the *unmerged* or *needs updating* state of a file during a merge if it was cleared by accident.

--info-only

Do not create objects in the object database for all <file> arguments that follow this flag; just insert their object IDs into the index.

--force-remove

Remove the file from the index even when the working directory still has such a file. (Implies --remove.)

--replace

By default, when a file *path* exists in the index, *git update-index* refuses an attempt to add *path/file*. Similarly if a file *path/file* exists, a file *path* cannot be added. With --replace flag, existing entries that conflict with the entry being added are automatically removed with warning messages.

--stdin

Instead of taking a list of paths from the command line, read a list of paths from the standard input. Paths are separated by LF (i.e. one path per line) by default.

--verbose

Report what is being added and removed from the index.

--index-version <n>

Write the resulting index out in the named on-disk format version. Supported versions are 2, 3, and 4. The current default version is 2 or 3, depending on whether extra features are used, such as *git add -N*. With *--verbose*, also report the version the index file uses before and after this command.

Version 4 performs a simple pathname compression that reduces index size by 30%-50% on large repositories, which results in faster load time. Git supports it since version 1.8.0, released in October 2012, and support for it was added to libgit2 in 2016 and to JGit in 2020. Older versions of this manual page called it "relatively young", but it should be considered mature technology these days.

--show-index-version

Report the index format version used by the on-disk index file. See *--index-version* above.

-z

Only meaningful with *--stdin* or *--index-info*; paths are separated with NUL character instead of LF.

--split-index , --no-split-index

Enable or disable split index mode. If split-index mode is already enabled and *--split-index* is given again, all changes in $GIT_DIR/index are pushed back to the shared index file.

These options take effect whatever the value of the *core.splitIndex* configuration variable (see Section G.3.29, “git-config(1)”). But a warning is emitted when the change goes against the configured value, as the configured value will take effect next time the index is read and this will remove the intended effect of the option.

--untracked-cache , --no-untracked-cache

Enable or disable untracked cache feature. Please use *--test-untracked-cache* before enabling it.

These options take effect whatever the value of the *core.untrackedCache* configuration variable (see Section G.3.29, “git-config(1)”). But a warning is emitted when the change goes against the configured value, as the configured value will take effect next time the index is read and this will remove the intended effect of the option.

--test-untracked-cache

Only perform tests on the working directory to make sure untracked cache can be used. You have to manually enable untracked cache using *--untracked-cache* or *--force-untracked-cache* or the *core.untrackedCache* configuration variable afterwards if you really want to use it. If a test fails the exit code is 1 and a message explains what is not working as needed, otherwise the exit code is 0 and OK is printed.

--force-untracked-cache

Same as *--untracked-cache*. Provided for backwards compatibility with older versions of Git where *--untracked-cache* used to imply *--test-untracked-cache* but this option would enable the extension unconditionally.

--fsmonitor , --no-fsmonitor

Enable or disable files system monitor feature. These options take effect whatever the value of the *core.fsmonitor* configuration variable (see Section G.3.29, “git-config(1)”). But a warning is emitted when the change goes against the configured value, as the configured value will take effect next time the index is read and this will remove the intended effect of the option.

--

Do not interpret any more arguments as options.

<file>

Files to act on. Note that files beginning with . are discarded. This includes *./file* and *dir/./file*. If you don't want this, then use cleaner names. The same applies to directories ending / and paths with //

USING --REFRESH

--refresh does not calculate a new sha1 file or bring the index up to date for mode/content changes. But what it **does** do is to "re-match" the stat information of a file with the index, so that you can refresh the index for a file that hasn't been changed but where the stat entry is out of date.

For example, you'd want to do this after doing a *git read-tree*, to link up the stat index details with the proper files.

USING --CACHEINFO OR --INFO-ONLY

--cacheinfo is used to register a file that is not in the current working directory. This is useful for minimum-checkout merging.

To pretend you have a file at path with mode and sha1, say:

```
$ git update-index --add --cacheinfo <mode>,<sha1>,<path>
```

--info-only is used to register files without placing them in the object database. This is useful for status-only repositories.

Both *--cacheinfo* and *--info-only* behave similarly: the index is updated but the object database isn't. *--cacheinfo* is useful when the object is in the database but the file isn't available locally. *--info-only* is useful when the file is available, but you do not wish to update the object database.

USING --INDEX-INFO

--index-info is a more powerful mechanism that lets you feed multiple entry definitions from the standard input, and designed specifically for scripts. It can take inputs of three formats:

1. mode SP type SP sha1 TAB path

   This format is to stuff *git ls-tree* output into the index.

2. mode SP sha1 SP stage TAB path

   This format is to put higher order stages into the index file and matches *git ls-files --stage* output.

3. mode SP sha1 TAB path

   This format is no longer produced by any Git command, but is and will continue to be supported by *update-index --index-info*.

To place a higher stage entry to the index, the path should first be removed by feeding a mode=0 entry for the path, and then feeding necessary input lines in the third format.

For example, starting with this index:

```
$ git ls-files -s
100644 8a1218a1024a212bb3db30becd860315f9f3ac52 0       frotz
```

you can feed the following input to *--index-info*:

```
$ git update-index --index-info
0 0000000000000000000000000000000000000000      frotz
100644 8a1218a1024a212bb3db30becd860315f9f3ac52 1       frotz
100755 8a1218a1024a212bb3db30becd860315f9f3ac52 2       frotz
```

The first line of the input feeds 0 as the mode to remove the path; the SHA-1 does not matter as long as it is well formatted. Then the second and third line feeds stage 1 and stage 2 entries for that path. After the above, we would end up with this:

```
$ git ls-files -s
100644 8a1218a1024a212bb3db30becd860315f9f3ac52 1       frotz
100755 8a1218a1024a212bb3db30becd860315f9f3ac52 2       frotz
```

USING ASSUME UNCHANGED BIT

Many operations in Git depend on your filesystem to have an efficient *lstat(2)* implementation, so that *st_mtime* information for working tree files can be cheaply checked to see if the file contents have changed from the version recorded in the index file. Unfortunately, some filesystems have inefficient *lstat(2)*. If your filesystem is one of them, you can set "assume unchanged" bit to paths you have not changed to cause Git not to do this check. Note that setting this bit on a path does not mean Git will check the contents of the file to see if it has changed -- it makes Git to omit any checking and assume it has **not** changed. When you make changes to working tree files, you have to explicitly tell Git about it by dropping "assume unchanged" bit, either before or after you modify them.

In order to set "assume unchanged" bit, use *--assume-unchanged* option. To unset, use *--no-assume-unchanged*. To see which files have the "assume unchanged" bit set, use *git ls-files -v* (see Section G.3.75, “git-ls-files(1)”).

The command looks at *core.ignorestat* configuration variable. When this is true, paths updated with *git update-index paths...* and paths updated with other Git commands that update both index and working tree (e.g. *git apply --index*, *git checkout-index -u*, and *git read-tree -u*) are automatically marked as "assume unchanged". Note that "assume unchanged" bit is **not** set if *git update-index --refresh* finds the working tree file matches the index (use *git update-index --really-refresh* if you want to mark them as "assume unchanged").

Sometimes users confuse the assume-unchanged bit with the skip-worktree bit. See the final paragraph in the "Skip-worktree bit" section below for an explanation of the differences.

EXAMPLES

To update and refresh only the files already checked out:

```
$ git checkout-index -n -f -a && git update-index --ignore-missing --
refresh
```

On an inefficient filesystem with *core.ignorestat* set

```
$ git update-index --really-refresh              ❶
$ git update-index --no-assume-unchanged foo.c   ❷
$ git diff --name-only                           ❸
$ edit foo.c
$ git diff --name-only                           ❹
M foo.c
$ git update-index foo.c                         ❺
$ git diff --name-only                           ❻
$ edit foo.c
$ git diff --name-only                           ❼
$ git update-index --no-assume-unchanged foo.c   ❽
$ git diff --name-only                           ❾
M foo.c
```

❶ forces lstat(2) to set "assume unchanged" bits for paths that match index.
❷ mark the path to be edited.
❸ this does lstat(2) and finds index matches the path.
❹ this does lstat(2) and finds index does **not** match the path.
❺ registering the new version to index sets "assume unchanged" bit.
❻ and it is assumed unchanged.
❼ even after you edit it.
❽ you can tell about the change after the fact.
❾ now it checks with lstat(2) and finds it has been changed.

SKIP-WORKTREE BIT

Skip-worktree bit can be defined in one (long) sentence: Tell git to avoid writing the file to the working directory when reasonably possible, and treat the file as unchanged when it is not present in the working directory.

Note that not all git commands will pay attention to this bit, and some only partially support it.

The update-index flags and the read-tree capabilities relating to the skip-worktree bit predated the introduction of the Section G.3.135, “git-sparse-checkout(1)” command, which provides a much easier way to configure and handle the skip-worktree bits. If you want to reduce your working tree to only deal with a subset of the files in the repository, we strongly encourage the use of Section G.3.135, “git-sparse-checkout(1)” in preference to the low-level update-index and read-tree primitives.

The primary purpose of the skip-worktree bit is to enable sparse checkouts, i.e. to have working directories with only a subset of paths present. When the skip-worktree bit is set, Git commands (such as *switch*, *pull*, *merge*) will avoid writing these files. However, these commands will sometimes write these files anyway in important cases such as conflicts during a merge or rebase. Git commands will also avoid treating the lack of such files as an intentional deletion; for example *git add -u* will not stage a deletion for these files and *git commit -a* will not make a commit deleting them either.

Although this bit looks similar to assume-unchanged bit, its goal is different. The assume-unchanged bit is for leaving the file in the working tree but having Git omit checking it for changes and presuming that the file has not been changed (though if it can determine without stat'ing the file that it has changed, it is free to record the changes). skip-worktree tells Git to ignore the absence of the file, avoid updating it when possible with commands that normally update much of the working directory (e.g. *checkout*, *switch*, *pull*, etc.), and not have its absence be recorded in commits. Note that in sparse checkouts (setup by *git sparse-checkout* or by configuring core.sparseCheckout to true), if a file is marked as skip-worktree in the index but is found in the working tree, Git will clear the skip-worktree bit for that file.

SPLIT INDEX

This mode is designed for repositories with very large indexes, and aims at reducing the time it takes to repeatedly write these indexes.

In this mode, the index is split into two files, $GIT_DIR/index and $GIT_DIR/sharedindex.<SHA-1>. Changes are accumulated in $GIT_DIR/index, the split index, while the shared index file contains all index entries and stays unchanged.

All changes in the split index are pushed back to the shared index file when the number of entries in the split index reaches a level specified by the splitIndex.maxPercentChange config variable (see Section G.3.29, “git-config(1)”).

Each time a new shared index file is created, the old shared index files are deleted if their modification time is older than what is specified by the splitIndex.sharedIndexExpire config variable (see Section G.3.29, “git-config(1)”).

To avoid deleting a shared index file that is still used, its modification time is updated to the current time every time a new split index based on the shared index file is either created or read from.

UNTRACKED CACHE

This cache is meant to speed up commands that involve determining untracked files such as *git status*.

This feature works by recording the mtime of the working tree directories and then omitting reading directories and stat calls against files in those directories whose mtime hasn't changed. For this to work the underlying operating system and file system must change the *st_mtime* field of directories if files in the directory are added, modified or deleted.

You can test whether the filesystem supports that with the *--test-untracked-cache* option. The *--untracked-cache* option used to implicitly perform that test in older versions of Git, but that's no longer the case.

If you want to enable (or disable) this feature, it is easier to use the *core.untrackedCache* configuration variable (see Section G.3.29, “git-config(1)”) than using the *--untracked-cache* option to *git update-index* in each repository, especially if you want to do so across all repositories you use, because you can set the configuration variable to *true* (or *false*) in your *$HOME/.gitconfig* just once and have it affect all repositories you touch.

When the *core.untrackedCache* configuration variable is changed, the untracked cache is added to or removed from the index the next time a command reads the index; while when *--[no-|force-]untracked-cache* are used, the untracked cache is immediately added to or removed from the index.

Before 2.17, the untracked cache had a bug where replacing a directory with a symlink to another directory could cause it to incorrectly show files tracked by git as untracked. See the "status: add a failing test showing a core.untrackedCache bug" commit to git.git. A workaround for that is (and this might work for other undiscovered bugs in the future):

```
$ git -c core.untrackedCache=false status
```

This bug has also been shown to affect non-symlink cases of replacing a directory with a file when it comes to the internal structures of the untracked cache, but no case has been reported where this resulted in wrong "git status" output.

There are also cases where existing indexes written by git versions before 2.17 will reference directories that don't exist anymore, potentially causing many "could not open directory" warnings to be printed on "git status". These are new warnings for existing issues that were previously silently discarded.

As with the bug described above the solution is to one-off do a "git status" run with *core.untrackedCache=false* to flush out the leftover bad data.

FILE SYSTEM MONITOR

This feature is intended to speed up git operations for repos that have large working directories.

It enables git to work together with a file system monitor (see Section G.3.57, “git-fsmonitor--daemon(1)” and the "fsmonitor-watchman" section of Section G.4.7, “githooks(5)”) that can inform it as to what files have been modified. This enables git to avoid having to lstat() every file to find modified files.

When used in conjunction with the untracked cache, it can further improve performance by avoiding the cost of scanning the entire working directory looking for new files.

If you want to enable (or disable) this feature, it is easier to use the *core.fsmonitor* configuration variable (see Section G.3.29, “git-config(1)”) than using the *--fsmonitor* option to *git update-index* in each repository, especially if you want to do so across all repositories you use, because you can set the configuration variable in your *$HOME/.gitconfig* just once and have it affect all repositories you touch.

When the *core.fsmonitor* configuration variable is changed, the file system monitor is added to or removed from the index the next time a command reads the index. When *--[no-]fsmonitor* are used, the file system monitor is immediately added to or removed from the index.

CONFIGURATION

The command honors *core.filemode* configuration variable. If your repository is on a filesystem whose executable bits are unreliable, this should be set to *false* (see Section G.3.29, “git-config(1)”). This causes the command to ignore differences in file modes recorded in the index and the file mode on the filesystem if they differ only on executable bit. On such an unfortunate filesystem, you may need to use *git update-index --chmod=*.

Quite similarly, if *core.symlinks* configuration variable is set to *false* (see Section G.3.29, “git-config(1)”), symbolic links are checked out as plain files, and this command does not modify a recorded file mode from symbolic link to regular file.

The command looks at *core.ignorestat* configuration variable. See *Using "assume unchanged" bit* section above.

The command also looks at *core.trustctime* configuration variable. It can be useful when the inode change time is regularly modified by something outside Git (file system crawlers and backup systems use ctime for marking files processed) (see Section G.3.29, “git-config(1)”).

The untracked cache extension can be enabled by the *core.untrackedCache* configuration variable (see Section G.3.29, “git-config(1)”).

NOTES

Users often try to use the assume-unchanged and skip-worktree bits to tell Git to ignore changes to files that are tracked. This does not work as expected, since Git may still check working tree files against the index when performing certain operations. In general, Git does not provide a way to ignore changes to tracked files, so alternate solutions are recommended.

For example, if the file you want to change is some sort of config file, the repository can include a sample config file that can then be copied into the ignored name and modified. The repository can even include a script to treat the sample file as a template, modifying and copying it automatically.

SEE ALSO

Section G.3.29, “git-config(1)”, Section G.3.2, “git-add(1)”, Section G.3.75, “git-ls-files(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.148. git-update-ref(1)

2024-07-02

Revision History

2024-07-02

NAME

git-update-ref - Update the object name stored in a ref safely

SYNOPSIS

git update-ref [-m <reason>] [--no-deref] (-d <ref> [<old-oid>] | [--create-reflog] <ref> <new-oid> [<old-oid>] | --stdin [-

DESCRIPTION

Given two arguments, stores the <new-oid> in the <ref>, possibly dereferencing the symbolic refs. E.g. *git update-ref HEAD <new-oid>* updates the current branch head to the new object.

Given three arguments, stores the <new-oid> in the <ref>, possibly dereferencing the symbolic refs, after verifying that the current value of the <ref> matches <old-oid>. E.g. *git update-ref refs/heads/master <new-oid> <old-oid>* updates the master branch head to <new-oid> only if its current value is <old-oid>. You can specify 40 "0" or an empty string as <old-oid> to make sure that the ref you are creating does not exist.

It also allows a "ref" file to be a symbolic pointer to another ref file by starting with the four-byte header sequence of "ref:".

More importantly, it allows the update of a ref file to follow these symbolic pointers, whether they are symlinks or these "regular file symbolic refs". It follows **real** symlinks only if they start with "refs/": otherwise it will just try to read them and update them as a regular file (i.e. it will allow the filesystem to follow them, but will overwrite such a symlink to somewhere else with a regular filename).

If --no-deref is given, <ref> itself is overwritten, rather than the result of following the symbolic pointers.

In general, using

```
git update-ref HEAD "$head"
```

should be a *lot* safer than doing

```
echo "$head" > "$GIT_DIR/HEAD"
```

both from a symlink following standpoint **and** an error checking standpoint. The "refs/" rule for symlinks means that symlinks that point to "outside" the tree are safe: they'll be followed for reading but not for writing (so we'll never write through a ref symlink to some other tree, if you have copied a whole archive by creating a symlink tree).

With *-d* flag, it deletes the named <ref> after verifying it still contains <old-oid>.

With *--stdin*, update-ref reads instructions from standard input and performs all modifications together. Specify commands of the form:

```
update SP <ref> SP <new-oid> [SP <old-oid>] LF
create SP <ref> SP <new-oid> LF
delete SP <ref> [SP <old-oid>] LF
verify SP <ref> [SP <old-oid>] LF
option SP <opt> LF
start LF
prepare LF
commit LF
abort LF
```

With *--create-reflog*, update-ref will create a reflog for each ref even if one would not ordinarily be created.

Quote fields containing whitespace as if they were strings in C source code; i.e., surrounded by double-quotes and with backslash escapes. Use 40 "0" characters or the empty string to specify a zero value. To specify a missing value, omit the value and its preceding SP entirely.

Alternatively, use *-z* to specify in NUL-terminated format, without quoting:

```
update SP <ref> NUL <new-oid> NUL [<old-oid>] NUL
create SP <ref> NUL <new-oid> NUL
```

```
delete SP <ref> NUL [<old-oid>] NUL
verify SP <ref> NUL [<old-oid>] NUL
option SP <opt> NUL
start NUL
prepare NUL
commit NUL
abort NUL
```

In this format, use 40 "0" to specify a zero value, and use the empty string to specify a missing value.

In either format, values can be specified in any form that Git recognizes as an object name. Commands in any other format or a repeated <ref> produce an error. Command meanings are:

update

Set <ref> to <new-oid> after verifying <old-oid>, if given. Specify a zero <new-oid> to ensure the ref does not exist after the update and/or a zero <old-oid> to make sure the ref does not exist before the update.

create

Create <ref> with <new-oid> after verifying it does not exist. The given <new-oid> may not be zero.

delete

Delete <ref> after verifying it exists with <old-oid>, if given. If given, <old-oid> may not be zero.

verify

Verify <ref> against <old-oid> but do not change it. If <old-oid> is zero or missing, the ref must not exist.

option

Modify the behavior of the next command naming a <ref>. The only valid option is *no-deref* to avoid dereferencing a symbolic ref.

start

Start a transaction. In contrast to a non-transactional session, a transaction will automatically abort if the session ends without an explicit commit. This command may create a new empty transaction when the current one has been committed or aborted already.

prepare

Prepare to commit the transaction. This will create lock files for all queued reference updates. If one reference could not be locked, the transaction will be aborted.

commit

Commit all reference updates queued for the transaction, ending the transaction.

abort

Abort the transaction, releasing all locks if the transaction is in prepared state.

If all <ref>s can be locked with matching <old-oid>s simultaneously, all modifications are performed. Otherwise, no modifications are performed. Note that while each individual <ref> is updated or deleted atomically, a concurrent reader may still see a subset of the modifications.

LOGGING UPDATES

If config parameter "core.logAllRefUpdates" is true and the ref is one under "refs/heads/", "refs/remotes/", "refs/notes/", or a pseudoref like HEAD or ORIG_HEAD; or the file "$GIT_DIR/logs/<ref>" exists then *git update-ref*

will append a line to the log file "$GIT_DIR/logs/<ref>" (dereferencing all symbolic refs before creating the log name) describing the change in ref value. Log lines are formatted as:

```
oldsha1 SP newsha1 SP committer LF
```

Where "oldsha1" is the 40 character hexadecimal value previously stored in <ref>, "newsha1" is the 40 character hexadecimal value of <new-oid> and "committer" is the committer's name, email address and date in the standard Git committer ident format.

Optionally with -m:

```
oldsha1 SP newsha1 SP committer TAB message LF
```

Where all fields are as described above and "message" is the value supplied to the -m option.

An update will fail (without changing <ref>) if the current user is unable to create a new log file, append to the existing log file or does not have committer information available.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.149. git-update-server-info(1)

2024-07-02

Revision History

2024-07-02

NAME

git-update-server-info - Update auxiliary info file to help dumb servers

SYNOPSIS

git update-server-info [-f | --force]

DESCRIPTION

A dumb server that does not do on-the-fly pack generations must have some auxiliary information files in $GIT_DIR/info and $GIT_OBJECT_DIRECTORY/info directories to help clients discover what references and packs the server has. This command generates such auxiliary files.

OPTIONS

-f , --force

Update the info files from scratch.

OUTPUT

Currently the command updates the following files. Please see Section G.4.13, “gitrepository-layout(5)” for a description of what they are for:

- objects/info/packs
- info/refs

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.150. git-upload-archive(1)

2024-07-02

Revision History

2024-07-02

NAME

git-upload-archive - Send archive back to git-archive

SYNOPSIS

git upload-archive <repository>

DESCRIPTION

Invoked by *git archive --remote* and sends a generated archive to the other end over the Git protocol.

This command is usually not invoked directly by the end user. The UI for the protocol is on the *git archive* side, and the program pair is meant to be used to get an archive from a remote repository.

SECURITY

In order to protect the privacy of objects that have been removed from history but may not yet have been pruned, *git-upload-archive* avoids serving archives for commits and trees that are not reachable from the repository's refs. However, because calculating object reachability is computationally expensive, *git-upload-archive* implements a stricter but easier-to-check set of rules:

1. Clients may request a commit or tree that is pointed to directly by a ref. E.g., *git archive --remote=origin v1.0*.
2. Clients may request a sub-tree within a commit or tree using the *ref:path* syntax. E.g., *git archive --remote=origin v1.0:Documentation*.
3. Clients may *not* use other sha1 expressions, even if the end result is reachable. E.g., neither a relative commit like *master^* nor a literal sha1 like *abcd1234* is allowed, even if the result is reachable from the refs.

Note that rule 3 disallows many cases that do not have any privacy implications. These rules are subject to change in future versions of git, and the server accessed by *git archive --remote* may or may not follow these exact rules.

If the config option *uploadArchive.allowUnreachable* is true, these rules are ignored, and clients may use arbitrary sha1 expressions. This is useful if you do not care about the privacy of unreachable objects, or if your object database is already publicly available for access via non-smart-http.

OPTIONS

<repository>

The repository to get a tar archive from.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.151. git-upload-pack(1)

2024-07-02

Revision History

2024-07-02

NAME

git-upload-pack - Send objects packed back to git-fetch-pack

SYNOPSIS

git-upload-pack [--[no-]strict] [--timeout=<n>] [--stateless-rpc]

[--advertise-refs] <directory>

DESCRIPTION

Invoked by *git fetch-pack*, learns what objects the other side is missing, and sends them after packing.

This command is usually not invoked directly by the end user. The UI for the protocol is on the *git fetch-pack* side, and the program pair is meant to be used to pull updates from a remote repository. For push operations, see *git send-pack*.

OPTIONS

--[no-]strict

Do not try <directory>/.git/ if <directory> is not a Git directory.

--timeout=<n>

Interrupt transfer after <n> seconds of inactivity.

--stateless-rpc

Perform only a single read-write cycle with stdin and stdout. This fits with the HTTP POST request processing model where a program may read the request, write a response, and must exit.

--http-backend-info-refs

Used by Section G.3.65, “git-http-backend(1)” to serve up *$GIT_URL/info/refs?service=git-upload-pack* requests. See "Smart Clients" in Section G.5.9, “gitprotocol-http(5)” and "HTTP Transport" in the Section G.5.11, “gitprotocol-v2(5)” documentation. Also understood by Section G.3.108, “git-receive-pack(1)”.

<directory>

The repository to sync from.

ENVIRONMENT

GIT_PROTOCOL

Internal variable used for handshaking the wire protocol. Server admins may need to configure some transports to allow this variable to be passed. See the discussion in Section G.3.1, “git(1)”.

GIT_NO_LAZY_FETCH

When cloning or fetching from a partial repository (i.e., one itself cloned with *--filter*), the server-side *upload-pack* may need to fetch extra objects from its upstream in order to complete the request. By default, *upload-pack* will refuse to perform such a lazy fetch, because *git fetch* may run arbitrary commands specified in configuration and hooks of the source repository (and *upload-pack* tries to be safe to run even in untrusted *.git* directories).

This is implemented by having *upload-pack* internally set the *GIT_NO_LAZY_FETCH* variable to *1*. If you want to override it (because you are fetching from a partial clone, and you are sure you trust it), you can explicitly set *GIT_NO_LAZY_FETCH* to *0*.

SECURITY

Most Git commands should not be run in an untrusted *.git* directory (see the section *SECURITY* in Section G.3.1, “git(1)”). *upload-pack* tries to avoid any dangerous configuration options or hooks from the repository it's serving, making it safe to clone an untrusted directory and run commands on the resulting clone.

For an extra level of safety, you may be able to run *upload-pack* as an alternate user. The details will be platform dependent, but on many systems you can run:

```
git clone --no-local --upload-pack='sudo -u nobody git-upload-pack' ...
```

SEE ALSO

Section G.4.11, “gitnamespaces(7)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.152. git-var(1)

2024-07-02
Revision History
2024-07-02

NAME

git-var - Show a Git logical variable

SYNOPSIS

git var (-l | <variable>)

DESCRIPTION

Prints a Git logical variable. Exits with code 1 if the variable has no value.

OPTIONS

-l

Display the logical variables. In addition, all the variables of the Git configuration file .git/config are listed as well. (However, the configuration variables listing functionality is deprecated in favor of *git config -l*.)

EXAMPLES

```
$ git var GIT_AUTHOR_IDENT
Eric W. Biederman <ebiederm@lnxi.com> 1121223278 -0600
```

VARIABLES

GIT_AUTHOR_IDENT

The author of a piece of code.

GIT_COMMITTER_IDENT

The person who put a piece of code into Git.

GIT_EDITOR

Text editor for use by Git commands. The value is meant to be interpreted by the shell when it is used. Examples: *~/bin/vi*, *$SOME_ENVIRONMENT_VARIABLE*, *"C:\Program Files\Vim\gvim.exe" --nofork*. The order of preference is the *$GIT_EDITOR* environment variable, then *core.editor* configuration, then *$VISUAL*, then *$EDITOR*, and then the default chosen at compile time, which is usually *vi*.

GIT_SEQUENCE_EDITOR

Text editor used to edit the *todo* file while running *git rebase -i*. Like *GIT_EDITOR*, the value is meant to be interpreted by the shell when it is used. The order of preference is the *$GIT_SEQUENCE_EDITOR* environment variable, then *sequence.editor* configuration, and then the value of *git var GIT_EDITOR*.

GIT_PAGER

Text viewer for use by Git commands (e.g., *less*). The value is meant to be interpreted by the shell. The order of preference is the *$GIT_PAGER* environment variable, then *core.pager* configuration, then *$PAGER*, and then the default chosen at compile time (usually *less*).

GIT_DEFAULT_BRANCH

The name of the first branch created in newly initialized repositories.

GIT_SHELL_PATH

The path of the binary providing the POSIX shell for commands which use the shell.

GIT_ATTR_SYSTEM

The path to the system Section G.4.2, “gitattributes(5)” file, if one is enabled.

GIT_ATTR_GLOBAL

The path to the global (per-user) Section G.4.2, “gitattributes(5)” file.

GIT_CONFIG_SYSTEM

The path to the system configuration file, if one is enabled.

GIT_CONFIG_GLOBAL

The path to the global (per-user) configuration files, if any.

Most path values contain only one value. However, some can contain multiple values, which are separated by newlines, and are listed in order from highest to lowest priority. Callers should be prepared for any such path value to contain multiple items.

Note that paths are printed even if they do not exist, but not if they are disabled by other environment variables.

SEE ALSO

Section G.3.27, “git-commit-tree(1)” Section G.3.144, “git-tag(1)” Section G.3.29, “git-config(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.153. git-verify-commit(1)

2024-07-02

Revision History

2024-07-02

NAME

git-verify-commit - Check the GPG signature of commits

SYNOPSIS

git verify-commit [-v | --verbose] [--raw] <commit>…

DESCRIPTION

Validates the GPG signature created by *git commit -S*.

OPTIONS

--raw

Print the raw gpg status output to standard error instead of the normal human-readable output.

-v , --verbose

Print the contents of the commit object before validating it.

<commit>…

SHA-1 identifiers of Git commit objects.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.154. git-verify-pack(1)

2024-07-02
Revision History
2024-07-02

NAME

git-verify-pack - Validate packed Git archive files

SYNOPSIS

git verify-pack [-v | --verbose] [-s | --stat-only] [--] <pack>.idx…

DESCRIPTION

Reads given idx file for packed Git archive created with the *git pack-objects* command and verifies the idx file and the corresponding pack file.

OPTIONS

<pack>.idx …

The idx files to verify.

-v , --verbose

After verifying the pack, show the list of objects contained in the pack and a histogram of delta chain length.

-s , --stat-only

Do not verify the pack contents; only show the histogram of delta chain length. With *--verbose*, the list of objects is also shown.

--

Do not interpret any more arguments as options.

OUTPUT FORMAT

When specifying the -v option the format used is:

```
SHA-1 type size size-in-packfile offset-in-packfile
```

for objects that are not deltified in the pack, and

```
SHA-1 type size size-in-packfile offset-in-packfile depth base-SHA-1
```

for objects that are deltified.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.155. git-verify-tag(1)

2024-07-02
Revision History
2024-07-02

NAME

git-verify-tag - Check the GPG signature of tags

SYNOPSIS

git verify-tag [-v | --verbose] [--format=<format>] [--raw] <tag>…

DESCRIPTION

Validates the gpg signature created by *git tag*.

OPTIONS

--raw

Print the raw gpg status output to standard error instead of the normal human-readable output.

-v , --verbose

Print the contents of the tag object before validating it.

<tag>…

SHA-1 identifiers of Git tag objects.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.156. git-version(1)

2024-07-02
Revision History
2024-07-02

NAME

git-version - Display version information about Git

SYNOPSIS

git version [--build-options]

DESCRIPTION

With no options given, the version of *git* is printed on the standard output.

Note that *git --version* is identical to *git version* because the former is internally converted into the latter.

OPTIONS

--build-options

Include additional information about how git was built for diagnostic purposes.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.157. git-web--browse(1)

2024-07-02
Revision History
2024-07-02

NAME

git-web--browse - Git helper script to launch a web browser

SYNOPSIS

git web--browse [<options>] (<URL>|<file>)…

DESCRIPTION

This script tries, as much as possible, to display the URLs and FILEs that are passed as arguments, as HTML pages in new tabs on an already opened web browser.

The following browsers (or commands) are currently supported:

- firefox (this is the default under X Window when not using KDE)
- iceweasel
- seamonkey
- iceape
- chromium (also supported as chromium-browser)
- google-chrome (also supported as chrome)
- konqueror (this is the default under KDE, see *Note about konqueror* below)
- opera
- w3m (this is the default outside graphical environments)
- elinks
- links
- lynx
- dillo
- open (this is the default under Mac OS X GUI)
- start (this is the default under MinGW)
- cygstart (this is the default under Cygwin)
- xdg-open

Custom commands may also be specified.

OPTIONS

-b <browser> , --browser=<browser>

Use the specified browser. It must be in the list of supported browsers.

-t <browser> , --tool=<browser>

Same as above.

-c <conf.var> , --config=<conf.var>

CONF.VAR is looked up in the Git config files. If it's set, then its value specifies the browser that should be used.

CONFIGURATION VARIABLES

1. CONF.VAR (from -c option) and web.browser

The web browser can be specified using a configuration variable passed with the -c (or --config) command-line option, or the *web.browser* configuration variable if the former is not used.

2. browser.<tool>.path

You can explicitly provide a full path to your preferred browser by setting the configuration variable *browser.<tool>.path*. For example, you can configure the absolute path to firefox by setting *browser.firefox.path*. Otherwise, *git web--browse* assumes the tool is available in PATH.

3. browser.<tool>.cmd

When the browser, specified by options or configuration variables, is not among the supported ones, then the corresponding *browser.<tool>.cmd* configuration variable will be looked up. If this variable exists then *git web--browse* will treat the specified tool as a custom command and will use a shell eval to run the command with the URLs passed as arguments.

NOTE ABOUT KONQUEROR

When *konqueror* is specified by a command-line option or a configuration variable, we launch *kfmclient* to try to open the HTML man page on an already opened konqueror in a new tab if possible.

For consistency, we also try such a trick if *browser.konqueror.path* is set to something like *A_PATH_TO/konqueror*. That means we will try to launch *A_PATH_TO/kfmclient* instead.

If you really want to use *konqueror*, then you can use something like the following:

```
[web]
        browser = konq

[browser "konq"]
        cmd = A_PATH_TO/konqueror
```

1. Note about git-config --global

Note that these configuration variables should probably be set using the *--global* flag, for example like this:

```
$ git config --global web.browser firefox
```

as they are probably more user specific than repository specific. See Section G.3.29, “git-config(1)” for more information about this.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.158. git-whatchanged(1)

2024-07-02
Revision History
2024-07-02

NAME

git-whatchanged - Show logs with differences each commit introduces

SYNOPSIS

git whatchanged <option>…

DESCRIPTION

Shows commit logs and diff output each commit introduces.

New users are encouraged to use Section G.3.74, “git-log(1)” instead. The *whatchanged* command is essentially the same as Section G.3.74, “git-log(1)” but defaults to showing the raw format diff output and skipping merges.

The command is primarily kept for historical reasons; fingers of many people who learned Git long before *git log* was invented by reading the Linux kernel mailing list are trained to type it.

Examples

git whatchanged -p v2.6.12.. include/scsi drivers/scsi

Show as patches the commits since version *v2.6.12* that changed any file in the include/scsi or drivers/scsi subdirectories

git whatchanged --since="2 weeks ago" -- gitk

Show the changes during the last two weeks to the file *gitk*. The "--" is necessary to avoid confusion with the **branch** named *gitk*

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.159. git-worktree(1)

2024-07-02
Revision History
2024-07-02

NAME

git-worktree - Manage multiple working trees

SYNOPSIS

git worktree add [-f] [--detach] [--checkout] [--lock [--reason <string>]]
[--orphan] [(-b | -B) <new-branch>] <path> [<commit-ish>]
git worktree list [-v | --porcelain [-z]]
git worktree lock [--reason <string>] <worktree>
git worktree move <worktree> <new-path>
git worktree prune [-n] [-v] [--expire <expire>]
git worktree remove [-f] <worktree>
git worktree repair [<path>…]

git worktree unlock <worktree>

DESCRIPTION

Manage multiple working trees attached to the same repository.

A git repository can support multiple working trees, allowing you to check out more than one branch at a time. With *git worktree add* a new working tree is associated with the repository, along with additional metadata that differentiates that working tree from others in the same repository. The working tree, along with this metadata, is called a "worktree".

This new worktree is called a "linked worktree" as opposed to the "main worktree" prepared by Section G.3.71, “git-init(1)” or Section G.3.24, “git-clone(1)”. A repository has one main worktree (if it's not a bare repository) and zero or more linked worktrees. When you are done with a linked worktree, remove it with *git worktree remove*.

In its simplest form, *git worktree add* <*path*> automatically creates a new branch whose name is the final component of <*path*>, which is convenient if you plan to work on a new topic. For instance, *git worktree add ../hotfix* creates new branch *hotfix* and checks it out at path *../hotfix*. To instead work on an existing branch in a new worktree, use *git worktree add* <*path*> <*branch*>. On the other hand, if you just plan to make some experimental changes or do testing without disturbing existing development, it is often convenient to create a *throwaway* worktree not associated with any branch. For instance, *git worktree add -d* <*path*> creates a new worktree with a detached *HEAD* at the same commit as the current branch.

If a working tree is deleted without using *git worktree remove*, then its associated administrative files, which reside in the repository (see "DETAILS" below), will eventually be removed automatically (see *gc.worktreePruneExpire* in Section G.3.29, “git-config(1)”), or you can run *git worktree prune* in the main or any linked worktree to clean up any stale administrative files.

If the working tree for a linked worktree is stored on a portable device or network share which is not always mounted, you can prevent its administrative files from being pruned by issuing the *git worktree lock* command, optionally specifying *--reason* to explain why the worktree is locked.

COMMANDS

add <path> [<commit-ish>]

Create a worktree at <*path*> and checkout <*commit-ish*> into it. The new worktree is linked to the current repository, sharing everything except per-worktree files such as *HEAD*, *index*, etc. As a convenience, <*commit-ish*> may be a bare "-", which is synonymous with *@{-1}*.

If <*commit-ish*> is a branch name (call it <*branch*>) and is not found, and neither *-b* nor *-B* nor *--detach* are used, but there does exist a tracking branch in exactly one remote (call it <*remote*>) with a matching name, treat as equivalent to:

```
$ git worktree add --track -b <branch> <path> <remote>/<branch>
```

If the branch exists in multiple remotes and one of them is named by the *checkout.defaultRemote* configuration variable, we'll use that one for the purposes of disambiguation, even if the <*branch*> isn't unique across all remotes. Set it to e.g. *checkout.defaultRemote=origin* to always checkout remote branches from there if <*branch*> is ambiguous but exists on the *origin* remote. See also *checkout.defaultRemote* in Section G.3.29, “git-config(1)”.

If <*commit-ish*> is omitted and neither *-b* nor *-B* nor *--detach* used, then, as a convenience, the new worktree is associated with a branch (call it <*branch*>) named after *$(basename* <*path*>*)*. If <*branch*> doesn't exist, a new branch based on *HEAD* is automatically created as if *-b* <*branch*> was given. If <*branch*> does exist, it will be checked out in the new worktree, if it's not checked out anywhere else, otherwise the command will refuse to create the worktree (unless *--force* is used).

If <*commit-ish*> is omitted, neither *--detach*, or *--orphan* is used, and there are no valid local branches (or remote branches if *--guess-remote* is specified) then, as a convenience, the new worktree is associated with a new unborn branch named <*branch*> (after *$(basename* <*path*>*)* if neither *-b* or *-B* is used) as if *--orphan*

was passed to the command. In the event the repository has a remote and *--guess-remote* is used, but no remote or local branches exist, then the command fails with a warning reminding the user to fetch from their remote first (or override by using *-f/--force*).

list

List details of each worktree. The main worktree is listed first, followed by each of the linked worktrees. The output details include whether the worktree is bare, the revision currently checked out, the branch currently checked out (or "detached HEAD" if none), "locked" if the worktree is locked, "prunable" if the worktree can be pruned by the *prune* command.

lock

If a worktree is on a portable device or network share which is not always mounted, lock it to prevent its administrative files from being pruned automatically. This also prevents it from being moved or deleted. Optionally, specify a reason for the lock with *--reason*.

move

Move a worktree to a new location. Note that the main worktree or linked worktrees containing submodules cannot be moved with this command. (The *git worktree repair* command, however, can reestablish the connection with linked worktrees if you move the main worktree manually.)

prune

Prune worktree information in *$GIT_DIR/worktrees*.

remove

Remove a worktree. Only clean worktrees (no untracked files and no modification in tracked files) can be removed. Unclean worktrees or ones with submodules can be removed with *--force*. The main worktree cannot be removed.

repair [<path>…]

Repair worktree administrative files, if possible, if they have become corrupted or outdated due to external factors.

For instance, if the main worktree (or bare repository) is moved, linked worktrees will be unable to locate it. Running *repair* in the main worktree will reestablish the connection from linked worktrees back to the main worktree.

Similarly, if the working tree for a linked worktree is moved without using *git worktree move*, the main worktree (or bare repository) will be unable to locate it. Running *repair* within the recently-moved worktree will reestablish the connection. If multiple linked worktrees are moved, running *repair* from any worktree with each tree's new *<path>* as an argument, will reestablish the connection to all the specified paths.

If both the main worktree and linked worktrees have been moved manually, then running *repair* in the main worktree and specifying the new *<path>* of each linked worktree will reestablish all connections in both directions.

unlock

Unlock a worktree, allowing it to be pruned, moved or deleted.

OPTIONS

-f , --force

By default, *add* refuses to create a new worktree when *<commit-ish>* is a branch name and is already checked out by another worktree, or if *<path>* is already assigned to some worktree but is missing (for instance, if *<path>* was deleted manually). This option overrides these safeguards. To add a missing but locked worktree path, specify *--force* twice.

move refuses to move a locked worktree unless *--force* is specified twice. If the destination is already assigned to some other worktree but is missing (for instance, if *<new-path>* was deleted manually), then *--force* allows the move to proceed; use *--force* twice if the destination is locked.

remove refuses to remove an unclean worktree unless *--force* is used. To remove a locked worktree, specify *--force* twice.

-b <new-branch> , -B <new-branch>

With *add*, create a new branch named *<new-branch>* starting at *<commit-ish>*, and check out *<new-branch>* into the new worktree. If *<commit-ish>* is omitted, it defaults to *HEAD*. By default, *-b* refuses to create a new branch if it already exists. *-B* overrides this safeguard, resetting *<new-branch>* to *<commit-ish>*.

-d , --detach

With *add*, detach *HEAD* in the new worktree. See "DETACHED HEAD" in Section G.3.19, “git-checkout(1)”.

--[no-]checkout

By default, *add* checks out *<commit-ish>*, however, *--no-checkout* can be used to suppress checkout in order to make customizations, such as configuring sparse-checkout. See "Sparse checkout" in Section G.3.106, “git-read-tree(1)”.

--[no-]guess-remote

With *worktree add <path>*, without *<commit-ish>*, instead of creating a new branch from *HEAD*, if there exists a tracking branch in exactly one remote matching the basename of *<path>*, base the new branch on the remote-tracking branch, and mark the remote-tracking branch as "upstream" from the new branch.

This can also be set up as the default behaviour by using the *worktree.guessRemote* config option.

--[no-]track

When creating a new branch, if *<commit-ish>* is a branch, mark it as "upstream" from the new branch. This is the default if *<commit-ish>* is a remote-tracking branch. See *--track* in Section G.3.10, “git-branch(1)” for details.

--lock

Keep the worktree locked after creation. This is the equivalent of *git worktree lock* after *git worktree add*, but without a race condition.

-n , --dry-run

With *prune*, do not remove anything; just report what it would remove.

--orphan

With *add*, make the new worktree and index empty, associating the worktree with a new unborn branch named *<new-branch>*.

--porcelain

With *list*, output in an easy-to-parse format for scripts. This format will remain stable across Git versions and regardless of user configuration. It is recommended to combine this with *-z*. See below for details.

-z

Terminate each line with a NUL rather than a newline when *--porcelain* is specified with *list*. This makes it possible to parse the output when a worktree path contains a newline character.

-q , --quiet

With *add*, suppress feedback messages.

-v , --verbose

With *prune*, report all removals.

With *list*, output additional information about worktrees (see below).

--expire <time>

With *prune*, only expire unused worktrees older than *<time>*.

With *list*, annotate missing worktrees as prunable if they are older than *<time>*.

--reason <string>

With *lock* or with *add --lock*, an explanation why the worktree is locked.

<worktree>

Worktrees can be identified by path, either relative or absolute.

If the last path components in the worktree's path is unique among worktrees, it can be used to identify a worktree. For example if you only have two worktrees, at */abc/def/ghi* and */abc/def/ggg*, then *ghi* or *def/ghi* is enough to point to the former worktree.

REFS

When using multiple worktrees, some refs are shared between all worktrees, but others are specific to an individual worktree. One example is *HEAD*, which is different for each worktree. This section is about the sharing rules and how to access refs of one worktree from another.

In general, all pseudo refs are per-worktree and all refs starting with *refs/* are shared. Pseudo refs are ones like *HEAD* which are directly under *$GIT_DIR* instead of inside *$GIT_DIR/refs*. There are exceptions, however: refs inside *refs/bisect*, *refs/worktree* and *refs/rewritten* are not shared.

Refs that are per-worktree can still be accessed from another worktree via two special paths, *main-worktree* and *worktrees*. The former gives access to per-worktree refs of the main worktree, while the latter to all linked worktrees.

For example, *main-worktree/HEAD* or *main-worktree/refs/bisect/good* resolve to the same value as the main worktree's *HEAD* and *refs/bisect/good* respectively. Similarly, *worktrees/foo/HEAD* or *worktrees/bar/refs/bisect/bad* are the same as *$GIT_COMMON_DIR/worktrees/foo/HEAD* and *$GIT_COMMON_DIR/worktrees/bar/refs/bisect/bad*.

To access refs, it's best not to look inside *$GIT_DIR* directly. Instead use commands such as Section G.3.121, “git-rev-parse(1)” or Section G.3.148, “git-update-ref(1)” which will handle refs correctly.

CONFIGURATION FILE

By default, the repository *config* file is shared across all worktrees. If the config variables *core.bare* or *core.worktree* are present in the common config file and *extensions.worktreeConfig* is disabled, then they will be applied to the main worktree only.

In order to have worktree-specific configuration, you can turn on the *worktreeConfig* extension, e.g.:

```
$ git config extensions.worktreeConfig true
```

In this mode, specific configuration stays in the path pointed by *git rev-parse --git-path config.worktree*. You can add or update configuration in this file with *git config --worktree*. Older Git versions will refuse to access repositories with this extension.

Note that in this file, the exception for *core.bare* and *core.worktree* is gone. If they exist in *$GIT_DIR/config*, you must move them to the *config.worktree* of the main worktree. You may also take this opportunity to review and move other configuration that you do not want to share to all worktrees:

- *core.worktree* should never be shared.
- *core.bare* should not be shared if the value is *core.bare=true*.
- *core.sparseCheckout* should not be shared, unless you are sure you always use sparse checkout for all worktrees.

See the documentation of *extensions.worktreeConfig* in Section G.3.29, “git-config(1)” for more details.

DETAILS

Each linked worktree has a private sub-directory in the repository's *$GIT_DIR/worktrees* directory. The private sub-directory's name is usually the base name of the linked worktree's path, possibly appended with a number to make it unique. For example, when *$GIT_DIR=/path/main/.git* the command *git worktree add /path/other/test-next next* creates the linked worktree in */path/other/test-next* and also creates a *$GIT_DIR/worktrees/test-next* directory (or *$GIT_DIR/worktrees/test-next1* if *test-next* is already taken).

Within a linked worktree, *$GIT_DIR* is set to point to this private directory (e.g. */path/main/.git/worktrees/test-next* in the example) and *$GIT_COMMON_DIR* is set to point back to the main worktree's *$GIT_DIR* (e.g. */path/main/.git*). These settings are made in a *.git* file located at the top directory of the linked worktree.

Path resolution via *git rev-parse --git-path* uses either *$GIT_DIR* or *$GIT_COMMON_DIR* depending on the path. For example, in the linked worktree *git rev-parse --git-path HEAD* returns */path/main/.git/worktrees/test-next/HEAD* (not */path/other/test-next/.git/HEAD* or */path/main/.git/HEAD*) while *git rev-parse --git-path refs/heads/master* uses *$GIT_COMMON_DIR* and returns */path/main/.git/refs/heads/master*, since refs are shared across all worktrees, except *refs/bisect*, *refs/worktree* and *refs/rewritten*.

See Section G.4.13, “gitrepository-layout(5)” for more information. The rule of thumb is do not make any assumption about whether a path belongs to *$GIT_DIR* or *$GIT_COMMON_DIR* when you need to directly access something inside *$GIT_DIR*. Use *git rev-parse --git-path* to get the final path.

If you manually move a linked worktree, you need to update the *gitdir* file in the entry's directory. For example, if a linked worktree is moved to */newpath/test-next* and its *.git* file points to */path/main/.git/worktrees/test-next*, then update */path/main/.git/worktrees/test-next/gitdir* to reference */newpath/test-next* instead. Better yet, run *git worktree repair* to reestablish the connection automatically.

To prevent a *$GIT_DIR/worktrees* entry from being pruned (which can be useful in some situations, such as when the entry's worktree is stored on a portable device), use the *git worktree lock* command, which adds a file named *locked* to the entry's directory. The file contains the reason in plain text. For example, if a linked worktree's *.git* file points to */path/main/.git/worktrees/test-next* then a file named */path/main/.git/worktrees/test-next/locked* will prevent the *test-next* entry from being pruned. See Section G.4.13, “gitrepository-layout(5)” for details.

When *extensions.worktreeConfig* is enabled, the config file *.git/worktrees/<id>/config.worktree* is read after *.git/config* is.

LIST OUTPUT FORMAT

The *worktree list* command has two output formats. The default format shows the details on a single line with columns. For example:

```
$ git worktree list
/path/to/bare-source            (bare)
/path/to/linked-worktree        abcd1234 [master]
/path/to/other-linked-worktree  1234abc  (detached HEAD)
```

The command also shows annotations for each worktree, according to its state. These annotations are:

- *locked*, if the worktree is locked.
- *prunable*, if the worktree can be pruned via *git worktree prune*.

```
$ git worktree list
```

```
/path/to/linked-worktree    abcd1234 [master]
/path/to/locked-worktree    acbd5678 (brancha) locked
/path/to/prunable-worktree  5678abc  (detached HEAD) prunable
```

For these annotations, a reason might also be available and this can be seen using the verbose mode. The annotation is then moved to the next line indented followed by the additional information.

```
$ git worktree list --verbose
/path/to/linked-worktree              abcd1234 [master]
/path/to/locked-worktree-no-reason    abcd5678 (detached HEAD) locked
/path/to/locked-worktree-with-reason  1234abcd (brancha)
        locked: worktree path is mounted on a portable device
/path/to/prunable-worktree            5678abc1 (detached HEAD)
        prunable: gitdir file points to non-existent location
```

Note that the annotation is moved to the next line if the additional information is available, otherwise it stays on the same line as the worktree itself.

1. Porcelain Format

The porcelain format has a line per attribute. If *-z* is given then the lines are terminated with NUL rather than a newline. Attributes are listed with a label and value separated by a single space. Boolean attributes (like *bare* and *detached*) are listed as a label only, and are present only if the value is true. Some attributes (like *locked*) can be listed as a label only or with a value depending upon whether a reason is available. The first attribute of a worktree is always *worktree*, an empty line indicates the end of the record. For example:

```
$ git worktree list --porcelain
worktree /path/to/bare-source
bare

worktree /path/to/linked-worktree
HEAD abcd1234abcd1234abcd1234abcd1234abcd1234
branch refs/heads/master

worktree /path/to/other-linked-worktree
HEAD 1234abc1234abc1234abc1234abc1234abc1234a
detached

worktree /path/to/linked-worktree-locked-no-reason
HEAD 5678abc5678abc5678abc5678abc5678abc5678c
branch refs/heads/locked-no-reason
locked

worktree /path/to/linked-worktree-locked-with-reason
HEAD 3456def3456def3456def3456def3456def3456b
branch refs/heads/locked-with-reason
locked reason why is locked

worktree /path/to/linked-worktree-prunable
HEAD 1233def1234def1234def1234def1234def1234b
detached
prunable gitdir file points to non-existent location
```

Unless *-z* is used any "unusual" characters in the lock reason such as newlines are escaped and the entire reason is quoted as explained for the configuration variable *core.quotePath* (see Section G.3.29, “git-config(1)”). For Example:

```
$ git worktree list --porcelain
...
```

```
locked "reason\nwhy is locked"
...
```

EXAMPLES

You are in the middle of a refactoring session and your boss comes in and demands that you fix something immediately. You might typically use Section G.3.137, “git-stash(1)” to store your changes away temporarily, however, your working tree is in such a state of disarray (with new, moved, and removed files, and other bits and pieces strewn around) that you don't want to risk disturbing any of it. Instead, you create a temporary linked worktree to make the emergency fix, remove it when done, and then resume your earlier refactoring session.

```
$ git worktree add -b emergency-fix ../temp master
$ pushd ../temp
# ... hack hack hack ...
$ git commit -a -m 'emergency fix for boss'
$ popd
$ git worktree remove ../temp
```

BUGS

Multiple checkout in general is still experimental, and the support for submodules is incomplete. It is NOT recommended to make multiple checkouts of a superproject.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.160. git-write-tree(1)

2024-07-02

Revision History

2024-07-02

NAME

git-write-tree - Create a tree object from the current index

SYNOPSIS

git write-tree [--missing-ok] [--prefix=<prefix>/]

DESCRIPTION

Creates a tree object using the current index. The name of the new tree object is printed to standard output.

The index must be in a fully merged state.

Conceptually, *git write-tree* sync()s the current index contents into a set of tree files. In order to have that match what is actually in your directory right now, you need to have done a *git update-index* phase before you did the *git write-tree*.

OPTIONS

--missing-ok

Normally *git write-tree* ensures that the objects referenced by the directory exist in the object database. This option disables this check.

--prefix=<prefix>/

Writes a tree object that represents a subdirectory <*prefix*>. This can be used to write the tree object for a subproject that is in the named subdirectory.

GIT

Part of the Section G.3.1, “git(1)” suite

G.3.161. scalar(1)

2024-07-02

Revision History

2024-07-02

NAME

scalar - A tool for managing large Git repositories

SYNOPSIS

```
scalar clone [--single-branch] [--branch <main-branch>] [--full-clone]
    [--[no-]src] <url> [<enlistment>]
scalar list
scalar register [<enlistment>]
scalar unregister [<enlistment>]
scalar run ( all | config | commit-graph | fetch | loose-objects | pack-files ) [<enlistment>]
scalar reconfigure [ --all | <enlistment> ]
scalar diagnose [<enlistment>]
scalar delete <enlistment>
```

DESCRIPTION

Scalar is a repository management tool that optimizes Git for use in large repositories. Scalar improves performance by configuring advanced Git settings, maintaining repositories in the background, and helping to reduce data sent across the network.

An important Scalar concept is the enlistment: this is the top-level directory of the project. It usually contains the subdirectory *src/* which is a Git worktree. This encourages the separation between tracked files (inside *src/*) and untracked files, such as build artifacts (outside *src/*). When registering an existing Git worktree with Scalar whose name is not *src*, the enlistment will be identical to the worktree.

The *scalar* command implements various subcommands, and different options depending on the subcommand. With the exception of *clone*, *list* and *reconfigure --all*, all subcommands expect to be run in an enlistment.

The following options can be specified *before* the subcommand:

-C <directory>

Before running the subcommand, change the working directory. This option imitates the same option of Section G.3.1, “git(1)”.

-c <key>=<value>

For the duration of running the specified subcommand, configure this setting. This option imitates the same option of Section G.3.1, “git(1)”.

COMMANDS

1. Clone

clone [<options>] <url> [<enlistment>]

Clones the specified repository, similar to Section G.3.24, “git-clone(1)”. By default, only commit and tree objects are cloned. Once finished, the worktree is located at *<enlistment>/src*.

The sparse-checkout feature is enabled (except when run with *--full-clone*) and the only files present are those in the top-level directory. Use *git sparse-checkout set* to expand the set of directories you want to see, or *git sparse-checkout disable* to expand to all files (see Section G.3.135, "git-sparse-checkout(1)" for more details). You can explore the subdirectories outside your sparse-checkout by using *git ls-tree HEAD[:<directory>]*.

-b <name> , --branch <name>

Instead of checking out the branch pointed to by the cloned repository's HEAD, check out the *<name>* branch instead.

--[no-]single-branch

Clone only the history leading to the tip of a single branch, either specified by the *--branch* option or the primary branch remote's *HEAD* points at.

Further fetches into the resulting repository will only update the remote-tracking branch for the branch this option was used for the initial cloning. If the HEAD at the remote did not point at any branch when *--single-branch* clone was made, no remote-tracking branch is created.

--[no-]src

By default, *scalar clone* places the cloned repository within a *<entlistment>/src* directory. Use *--no-src* to place the cloned repository directly in the *<enlistment>* directory.

--[no-]full-clone

A sparse-checkout is initialized by default. This behavior can be turned off via *--full-clone*.

2. List

list

List enlistments that are currently registered by Scalar. This subcommand does not need to be run inside an enlistment.

3. Register

register [<enlistment>]

Adds the enlistment's repository to the list of registered repositories and starts background maintenance. If *<enlistment>* is not provided, then the enlistment associated with the current working directory is registered.

Note: when this subcommand is called in a worktree that is called *src/*, its parent directory is considered to be the Scalar enlistment. If the worktree is *not* called *src/*, it itself will be considered to be the Scalar enlistment.

4. Unregister

unregister [<enlistment>]

Remove the specified repository from the list of repositories registered with Scalar and stop the scheduled background maintenance.

5. Run

scalar run (all | config | commit-graph | fetch | loose-objects | pack-files) [<enlistment>]

Run the given maintenance task (or all tasks, if *all* was specified). Except for *all* and *config*, this subcommand simply hands off to Section G.3.80, "git-maintenance(1)" (mapping *fetch* to *prefetch* and *pack-files* to *incremental-repack*).

These tasks are run automatically as part of the scheduled maintenance, as soon as the repository is registered with Scalar. It should therefore not be necessary to run this subcommand manually.

The *config* task is specific to Scalar and configures all those opinionated default settings that make Git work more efficiently with large repositories. As this task is run as part of *scalar clone* automatically, explicit invocations of this task are rarely needed.

6. Reconfigure

After a Scalar upgrade, or when the configuration of a Scalar enlistment was somehow corrupted or changed by mistake, this subcommand allows to reconfigure the enlistment.

With the *--all* option, all enlistments currently registered with Scalar will be reconfigured. Use this option after each Scalar upgrade.

7. Diagnose

diagnose [<enlistment>]

When reporting issues with Scalar, it is often helpful to provide the information gathered by this command, including logs and certain statistics describing the data shape of the current enlistment.

The output of this command is a *.zip* file that is written into a directory adjacent to the worktree in the *src* directory.

8. Delete

delete <enlistment>

This subcommand lets you delete an existing Scalar enlistment from your local file system, unregistering the repository.

SEE ALSO

Section G.3.24, “git-clone(1)”, Section G.3.80, “git-maintenance(1)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.4. Misc

G.4.1. gitcli(7)

2024-07-02

Revision History

2024-07-02

NAME

gitcli - Git command-line interface and conventions

SYNOPSIS

gitcli

DESCRIPTION

This manual describes the convention used throughout Git CLI.

Many commands take revisions (most often "commits", but sometimes "tree-ish", depending on the context and command) and paths as their arguments. Here are the rules:

- Options come first and then args. A subcommand may take dashed options (which may take their own arguments, e.g. "--max-parents 2") and arguments. You SHOULD give dashed options first and then arguments. Some commands may accept dashed options after you have already given non-option arguments (which may make the command ambiguous), but you should not rely on it (because eventually we may find a way to fix these ambiguities by enforcing the "options then args" rule).

- Revisions come first and then paths. E.g. in *git diff v1.0 v2.0 arch/x86 include/asm-x86*, *v1.0* and *v2.0* are revisions and *arch/x86* and *include/asm-x86* are paths.

- When an argument can be misunderstood as either a revision or a path, they can be disambiguated by placing -- between them. E.g. *git diff -- HEAD* is, "I have a file called HEAD in my work tree. Please show changes between the version I staged in the index and what I have in the work tree for that file", not "show the difference between the HEAD commit and the work tree as a whole". You can say *git diff HEAD* -- to ask for the latter.

- Without disambiguating --, Git makes a reasonable guess, but errors out and asks you to disambiguate when ambiguous. E.g. if you have a file called HEAD in your work tree, *git diff HEAD* is ambiguous, and you have to say either *git diff HEAD* -- or *git diff -- HEAD* to disambiguate.

- Because -- disambiguates revisions and paths in some commands, it cannot be used for those commands to separate options and revisions. You can use *--end-of-options* for this (it also works for commands that do not distinguish between revisions in paths, in which case it is simply an alias for --).

  When writing a script that is expected to handle random user-input, it is a good practice to make it explicit which arguments are which by placing disambiguating -- at appropriate places.

- Many commands allow wildcards in paths, but you need to protect them from getting globbed by the shell. These two mean different things:

  ```
  $ git restore *.c
  $ git restore \*.c
  ```

  The former lets your shell expand the fileglob, and you are asking the dot-C files in your working tree to be overwritten with the version in the index. The latter passes the **.c* to Git, and you are asking the paths in the index that match the pattern to be checked out to your working tree. After running *git add hello.c; rm hello.c*, you will *not* see *hello.c* in your working tree with the former, but with the latter you will.

- Just as the filesystem . (period) refers to the current directory, using a . as a repository name in Git (a dot-repository) is a relative path and means your current repository.

Here are the rules regarding the "flags" that you should follow when you are scripting Git:

- Splitting short options to separate words (prefer *git foo -a -b* to *git foo -ab*, the latter may not even work).

- When a command-line option takes an argument, use the *stuck* form. In other words, write *git foo -oArg* instead of *git foo -o Arg* for short options, and *git foo --long-opt=Arg* instead of *git foo --long-opt Arg* for long options. An option that takes optional option-argument must be written in the *stuck* form.

- When you give a revision parameter to a command, make sure the parameter is not ambiguous with a name of a file in the work tree. E.g. do not write *git log -1 HEAD* but write *git log -1 HEAD* --; the former will not work if you happen to have a file called *HEAD* in the work tree.

- Many commands allow a long option *--option* to be abbreviated only to their unique prefix (e.g. if there is no other option whose name begins with *opt*, you may be able to spell *--opt* to invoke the *--option* flag), but you should fully spell them out when writing your scripts; later versions of Git may introduce a new option whose name shares the same prefix, e.g. *--optimize*, to make a short prefix that used to be unique no longer unique.

ENHANCED OPTION PARSER

From the Git 1.5.4 series and further, many Git commands (not all of them at the time of the writing though) come with an enhanced option parser.

Here is a list of the facilities provided by this option parser.

1. Magic Options

Commands which have the enhanced option parser activated all understand a couple of magic command-line options:

-h

gives a pretty printed usage of the command.

```
$ git describe -h
usage: git describe [<options>] <commit-ish>*
   or: git describe [<options>] --dirty

    --contains            find the tag that comes after the commit
    --debug               debug search strategy on stderr
    --all                 use any ref
    --tags                use any tag, even unannotated
    --long                always use long format
    --abbrev[=<n>]        use <n> digits to display SHA-1s
```

Note that some subcommand (e.g. *git grep*) may behave differently when there are things on the command line other than *-h*, but *git subcmd -h* without anything else on the command line is meant to consistently give the usage.

--help-all

Some Git commands take options that are only used for plumbing or that are deprecated, and such options are hidden from the default usage. This option gives the full list of options.

2. Negating options

Options with long option names can be negated by prefixing *--no-*. For example, *git branch* has the option *--track* which is *on* by default. You can use *--no-track* to override that behaviour. The same goes for *--color* and *--no-color*.

3. Aggregating short options

Commands that support the enhanced option parser allow you to aggregate short options. This means that you can for example use *git rm -rf* or *git clean -fdx*.

4. Abbreviating long options

Commands that support the enhanced option parser accepts unique prefix of a long option as if it is fully spelled out, but use this with a caution. For example, *git commit --amen* behaves as if you typed *git commit --amend*, but that is true only until a later version of Git introduces another option that shares the same prefix, e.g. *git commit --amenity* option.

5. Separating argument from the option

You can write the mandatory option parameter to an option as a separate word on the command line. That means that all the following uses work:

```
$ git foo --long-opt=Arg
$ git foo --long-opt Arg
$ git foo -oArg
$ git foo -o Arg
```

However, this is **NOT** allowed for switches with an optional value, where the *stuck* form must be used:

```
$ git describe --abbrev HEAD     # correct
$ git describe --abbrev=10 HEAD  # correct
```

```
$ git describe --abbrev 10 HEAD  # NOT WHAT YOU MEANT
```

NOTES ON FREQUENTLY CONFUSED OPTIONS

Many commands that can work on files in the working tree and/or in the index can take *--cached* and/or *--index* options. Sometimes people incorrectly think that, because the index was originally called cache, these two are synonyms. They are **not** -- these two options mean very different things.

- The *--cached* option is used to ask a command that usually works on files in the working tree to **only** work with the index. For example, *git grep*, when used without a commit to specify from which commit to look for strings in, usually works on files in the working tree, but with the *--cached* option, it looks for strings in the index.
- The *--index* option is used to ask a command that usually works on files in the working tree to **also** affect the index. For example, *git stash apply* usually merges changes recorded in a stash entry to the working tree, but with the *--index* option, it also merges changes to the index as well.

git apply command can be used with *--cached* and *--index* (but not at the same time). Usually the command only affects the files in the working tree, but with *--index*, it patches both the files and their index entries, and with *--cached*, it modifies only the index entries.

See also https://lore.kernel.org/git/7v64clg5u9.fsf@assigned-by-dhcp.cox.net/ and https://lore.kernel.org/git/7vy7ej9g38.fsf@gitster.siamese.dyndns.org/ for further information.

Some other commands that also work on files in the working tree and/or in the index can take *--staged* and/or *--worktree*.

- *--staged* is exactly like *--cached*, which is used to ask a command to only work on the index, not the working tree.
- *--worktree* is the opposite, to ask a command to work on the working tree only, not the index.
- The two options can be specified together to ask a command to work on both the index and the working tree.

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.2. gitattributes(5)

2024-07-02

Revision History

2024-07-02

NAME

gitattributes - Defining attributes per path

SYNOPSIS

$GIT_DIR/info/attributes, .gitattributes

DESCRIPTION

A *gitattributes* file is a simple text file that gives *attributes* to pathnames.

Each line in *gitattributes* file is of form:

```
pattern attr1 attr2 ...
```

That is, a pattern followed by an attributes list, separated by whitespaces. Leading and trailing whitespaces are ignored. Lines that begin with # are ignored. Patterns that begin with a double quote are quoted in C style. When the pattern matches the path in question, the attributes listed on the line are given to the path.

Each attribute can be in one of these states for a given path:

Set

The path has the attribute with special value "true"; this is specified by listing only the name of the attribute in the attribute list.

Unset

The path has the attribute with special value "false"; this is specified by listing the name of the attribute prefixed with a dash - in the attribute list.

Set to a value

The path has the attribute with specified string value; this is specified by listing the name of the attribute followed by an equal sign = and its value in the attribute list.

Unspecified

No pattern matches the path, and nothing says if the path has or does not have the attribute, the attribute for the path is said to be Unspecified.

When more than one pattern matches the path, a later line overrides an earlier line. This overriding is done per attribute.

The rules by which the pattern matches paths are the same as in *.gitignore* files (see Section G.4.5, “gitignore(5)”), with a few exceptions:

- negative patterns are forbidden
- patterns that match a directory do not recursively match paths inside that directory (so using the trailing-slash *path/* syntax is pointless in an attributes file; use *path/*** instead)

When deciding what attributes are assigned to a path, Git consults *$GIT_DIR/info/attributes* file (which has the highest precedence), *.gitattributes* file in the same directory as the path in question, and its parent directories up to the toplevel of the work tree (the further the directory that contains *.gitattributes* is from the path in question, the lower its precedence). Finally global and system-wide files are considered (they have the lowest precedence).

When the *.gitattributes* file is missing from the work tree, the path in the index is used as a fall-back. During checkout process, *.gitattributes* in the index is used and then the file in the working tree is used as a fall-back.

If you wish to affect only a single repository (i.e., to assign attributes to files that are particular to one user's workflow for that repository), then attributes should be placed in the *$GIT_DIR/info/attributes* file. Attributes which should be version-controlled and distributed to other repositories (i.e., attributes of interest to all users) should go into *.gitattributes* files. Attributes that should affect all repositories for a single user should be placed in a file specified by the *core.attributesFile* configuration option (see Section G.3.29, “git-config(1)”). Its default value is $XDG_CONFIG_HOME/git/attributes. If $XDG_CONFIG_HOME is either not set or empty, $HOME/.config/git/attributes is used instead. Attributes for all users on a system should be placed in the *$(prefix)/etc/gitattributes* file.

Sometimes you would need to override a setting of an attribute for a path to *Unspecified* state. This can be done by listing the name of the attribute prefixed with an exclamation point *!*.

RESERVED BUILTIN_* ATTRIBUTES

builtin_* is a reserved namespace for builtin attribute values. Any user defined attributes under this namespace will be ignored and trigger a warning.

1. *builtin_objectmode*

This attribute is for filtering files by their file bit modes (40000, 120000, 160000, 100755, 100644). e.g. *:(attr:builtin_objectmode=160000)*. You may also check these values with *git check-attr builtin_objectmode --<file>*. If the object is not in the index *git check-attr --cached* will return unspecified.

EFFECTS

Certain operations by Git can be influenced by assigning particular attributes to a path. Currently, the following operations are attributes-aware.

1. Checking-out and checking-in

These attributes affect how the contents stored in the repository are copied to the working tree files when commands such as *git switch*, *git checkout* and *git merge* run. They also affect how Git stores the contents you prepare in the working tree in the repository upon *git add* and *git commit*.

1.1. *text*

This attribute marks the path as a text file, which enables end-of-line conversion: When a matching file is added to the index, the file's line endings are normalized to LF in the index. Conversely, when the file is copied from the index to the working directory, its line endings may be converted from LF to CRLF depending on the *eol* attribute, the Git config, and the platform (see explanation of *eol* below).

Set

Setting the *text* attribute on a path enables end-of-line conversion on checkin and checkout as described above. Line endings are normalized to LF in the index every time the file is checked in, even if the file was previously added to Git with CRLF line endings.

Unset

Unsetting the *text* attribute on a path tells Git not to attempt any end-of-line conversion upon checkin or checkout.

Set to string value "auto"

When *text* is set to "auto", Git decides by itself whether the file is text or binary. If it is text and the file was not already in Git with CRLF endings, line endings are converted on checkin and checkout as described above. Otherwise, no conversion is done on checkin or checkout.

Unspecified

If the *text* attribute is unspecified, Git uses the *core.autocrlf* configuration variable to determine if the file should be converted.

Any other value causes Git to act as if *text* has been left unspecified.

1.2. *eol*

This attribute marks a path to use a specific line-ending style in the working tree when it is checked out. It has effect only if *text* or *text=auto* is set (see above), but specifying *eol* automatically sets *text* if *text* was left unspecified.

Set to string value "crlf"

This setting converts the file's line endings in the working directory to CRLF when the file is checked out.

Set to string value "lf"

This setting uses the same line endings in the working directory as in the index when the file is checked out.

Unspecified

If the *eol* attribute is unspecified for a file, its line endings in the working directory are determined by the *core.autocrlf* or *core.eol* configuration variable (see the definitions of those options in Section G.3.29, “git-config(1)”). If *text* is set but neither of those variables is, the default is *eol=crlf* on Windows and *eol=lf* on all other platforms.

1.3. Backwards compatibility with *crlf* attribute

For backwards compatibility, the *crlf* attribute is interpreted as follows:

```
crlf            text
-crlf           -text
crlf=input      eol=lf
```

1.4. End-of-line conversion

While Git normally leaves file contents alone, it can be configured to normalize line endings to LF in the repository and, optionally, to convert them to CRLF when files are checked out.

If you simply want to have CRLF line endings in your working directory regardless of the repository you are working with, you can set the config variable "core.autocrlf" without using any attributes.

```
[core]
        autocrlf = true
```

This does not force normalization of text files, but does ensure that text files that you introduce to the repository have their line endings normalized to LF when they are added, and that files that are already normalized in the repository stay normalized.

If you want to ensure that text files that any contributor introduces to the repository have their line endings normalized, you can set the *text* attribute to "auto" for *all* files.

```
*       text=auto
```

The attributes allow a fine-grained control, how the line endings are converted. Here is an example that will make Git normalize .txt, .vcproj and .sh files, ensure that .vcproj files have CRLF and .sh files have LF in the working directory, and prevent .jpg files from being normalized regardless of their content.

```
*               text=auto
*.txt           text
*.vcproj        text eol=crlf
*.sh            text eol=lf
*.jpg           -text
```

> **Note**
>
> When *text=auto* conversion is enabled in a cross-platform project using push and pull to a central repository the text files containing CRLFs should be normalized.

From a clean working directory:

```
$ echo "* text=auto" >.gitattributes
$ git add --renormalize .
$ git status        # Show files that will be normalized
$ git commit -m "Introduce end-of-line normalization"
```

If any files that should not be normalized show up in *git status*, unset their *text* attribute before running *git add -u*.

```
manual.pdf      -text
```

Conversely, text files that Git does not detect can have normalization enabled manually.

```
weirdchars.txt  text
```

If *core.safecrlf* is set to "true" or "warn", Git verifies if the conversion is reversible for the current setting of *core.autocrlf*. For "true", Git rejects irreversible conversions; for "warn", Git only prints a warning but accepts

an irreversible conversion. The safety triggers to prevent such a conversion done to the files in the work tree, but there are a few exceptions. Even though…

- *git add* itself does not touch the files in the work tree, the next checkout would, so the safety triggers;
- *git apply* to update a text file with a patch does touch the files in the work tree, but the operation is about text files and CRLF conversion is about fixing the line ending inconsistencies, so the safety does not trigger;
- *git diff* itself does not touch the files in the work tree, it is often run to inspect the changes you intend to next *git add*. To catch potential problems early, safety triggers.

1.5. *working-tree-encoding*

Git recognizes files encoded in ASCII or one of its supersets (e.g. UTF-8, ISO-8859-1, …) as text files. Files encoded in certain other encodings (e.g. UTF-16) are interpreted as binary and consequently built-in Git text processing tools (e.g. *git diff*) as well as most Git web front ends do not visualize the contents of these files by default.

In these cases you can tell Git the encoding of a file in the working directory with the *working-tree-encoding* attribute. If a file with this attribute is added to Git, then Git re-encodes the content from the specified encoding to UTF-8. Finally, Git stores the UTF-8 encoded content in its internal data structure (called "the index"). On checkout the content is re-encoded back to the specified encoding.

Please note that using the *working-tree-encoding* attribute may have a number of pitfalls:

- Alternative Git implementations (e.g. JGit or libgit2) and older Git versions (as of March 2018) do not support the *working-tree-encoding* attribute. If you decide to use the *working-tree-encoding* attribute in your repository, then it is strongly recommended to ensure that all clients working with the repository support it.

  For example, Microsoft Visual Studio resources files (**.rc*) or PowerShell script files (**.ps1*) are sometimes encoded in UTF-16. If you declare **.ps1* as files as UTF-16 and you add *foo.ps1* with a *working-tree-encoding* enabled Git client, then *foo.ps1* will be stored as UTF-8 internally. A client without *working-tree-encoding* support will checkout *foo.ps1* as UTF-8 encoded file. This will typically cause trouble for the users of this file.

  If a Git client that does not support the *working-tree-encoding* attribute adds a new file *bar.ps1*, then *bar.ps1* will be stored "as-is" internally (in this example probably as UTF-16). A client with *working-tree-encoding* support will interpret the internal contents as UTF-8 and try to convert it to UTF-16 on checkout. That operation will fail and cause an error.

- Reencoding content to non-UTF encodings can cause errors as the conversion might not be UTF-8 round trip safe. If you suspect your encoding to not be round trip safe, then add it to *core.checkRoundtripEncoding* to make Git check the round trip encoding (see Section G.3.29, “git-config(1)”). SHIFT-JIS (Japanese character set) is known to have round trip issues with UTF-8 and is checked by default.
- Reencoding content requires resources that might slow down certain Git operations (e.g *git checkout* or *git add*).

Use the *working-tree-encoding* attribute only if you cannot store a file in UTF-8 encoding and if you want Git to be able to process the content as text.

As an example, use the following attributes if your **.ps1* files are UTF-16 encoded with byte order mark (BOM) and you want Git to perform automatic line ending conversion based on your platform.

```
*.ps1            text working-tree-encoding=UTF-16
```

Use the following attributes if your **.ps1* files are UTF-16 little endian encoded without BOM and you want Git to use Windows line endings in the working directory (use *UTF-16LE-BOM* instead of *UTF-16LE* if you want UTF-16 little endian with BOM). Please note, it is highly recommended to explicitly define the line endings with *eol* if the *working-tree-encoding* attribute is used to avoid ambiguity.

```
*.ps1            text working-tree-encoding=UTF-16LE eol=CRLF
```

You can get a list of all available encodings on your platform with the following command:

```
iconv --list
```

If you do not know the encoding of a file, then you can use the *file* command to guess the encoding:

```
file foo.ps1
```

1.6. *ident*

When the attribute *ident* is set for a path, Git replaces *Id* in the blob object with *$Id:*, followed by the 40-character hexadecimal blob object name, followed by a dollar sign *$* upon checkout. Any byte sequence that begins with *$Id:* and ends with *$* in the worktree file is replaced with *Id* upon check-in.

1.7. *filter*

A *filter* attribute can be set to a string value that names a filter driver specified in the configuration.

A filter driver consists of a *clean* command and a *smudge* command, either of which can be left unspecified. Upon checkout, when the *smudge* command is specified, the command is fed the blob object from its standard input, and its standard output is used to update the worktree file. Similarly, the *clean* command is used to convert the contents of worktree file upon checkin. By default these commands process only a single blob and terminate. If a long running *process* filter is used in place of *clean* and/or *smudge* filters, then Git can process all blobs with a single filter command invocation for the entire life of a single Git command, for example *git add --all*. If a long running *process* filter is configured then it always takes precedence over a configured single blob filter. See section below for the description of the protocol used to communicate with a *process* filter.

One use of the content filtering is to massage the content into a shape that is more convenient for the platform, filesystem, and the user to use. For this mode of operation, the key phrase here is "more convenient" and not "turning something unusable into usable". In other words, the intent is that if someone unsets the filter driver definition, or does not have the appropriate filter program, the project should still be usable.

Another use of the content filtering is to store the content that cannot be directly used in the repository (e.g. a UUID that refers to the true content stored outside Git, or an encrypted content) and turn it into a usable form upon checkout (e.g. download the external content, or decrypt the encrypted content).

These two filters behave differently, and by default, a filter is taken as the former, massaging the contents into more convenient shape. A missing filter driver definition in the config, or a filter driver that exits with a non-zero status, is not an error but makes the filter a no-op passthru.

You can declare that a filter turns a content that by itself is unusable into a usable content by setting the filter.<driver>.required configuration variable to *true*.

Note: Whenever the clean filter is changed, the repo should be renormalized: $ git add --renormalize .

For example, in .gitattributes, you would assign the *filter* attribute for paths.

```
*.c     filter=indent
```

Then you would define a "filter.indent.clean" and "filter.indent.smudge" configuration in your .git/config to specify a pair of commands to modify the contents of C programs when the source files are checked in ("clean" is run) and checked out (no change is made because the command is "cat").

```
[filter "indent"]
        clean = indent
        smudge = cat
```

For best results, *clean* should not alter its output further if it is run twice ("clean#clean" should be equivalent to "clean"), and multiple *smudge* commands should not alter *clean*'s output ("smudge#smudge#clean" should be equivalent to "clean"). See the section on merging below.

The "indent" filter is well-behaved in this regard: it will not modify input that is already correctly indented. In this case, the lack of a smudge filter means that the clean filter *must* accept its own output without modifying it.

If a filter *must* succeed in order to make the stored contents usable, you can declare that the filter is *required*, in the configuration:

```
[filter "crypt"]
        clean = openssl enc ...
        smudge = openssl enc -d ...
        required
```

Sequence "%f" on the filter command line is replaced with the name of the file the filter is working on. A filter might use this in keyword substitution. For example:

```
[filter "p4"]
        clean = git-p4-filter --clean %f
        smudge = git-p4-filter --smudge %f
```

Note that "%f" is the name of the path that is being worked on. Depending on the version that is being filtered, the corresponding file on disk may not exist, or may have different contents. So, smudge and clean commands should not try to access the file on disk, but only act as filters on the content provided to them on standard input.

1.8. Long Running Filter Process

If the filter command (a string value) is defined via *filter.<driver>.process* then Git can process all blobs with a single filter invocation for the entire life of a single Git command. This is achieved by using the long-running process protocol (described in technical/long-running-process-protocol.txt).

When Git encounters the first file that needs to be cleaned or smudged, it starts the filter and performs the handshake. In the handshake, the welcome message sent by Git is "git-filter-client", only version 2 is supported, and the supported capabilities are "clean", "smudge", and "delay".

Afterwards Git sends a list of "key=value" pairs terminated with a flush packet. The list will contain at least the filter command (based on the supported capabilities) and the pathname of the file to filter relative to the repository root. Right after the flush packet Git sends the content split in zero or more pkt-line packets and a flush packet to terminate content. Please note, that the filter must not send any response before it received the content and the final flush packet. Also note that the "value" of a "key=value" pair can contain the "=" character whereas the key would never contain that character.

```
packet:          git> command=smudge
packet:          git> pathname=path/testfile.dat
packet:          git> 0000
packet:          git> CONTENT
packet:          git> 0000
```

The filter is expected to respond with a list of "key=value" pairs terminated with a flush packet. If the filter does not experience problems then the list must contain a "success" status. Right after these packets the filter is expected to send the content in zero or more pkt-line packets and a flush packet at the end. Finally, a second list of "key=value" pairs terminated with a flush packet is expected. The filter can change the status in the second list or keep the status as is with an empty list. Please note that the empty list must be terminated with a flush packet regardless.

```
packet:          git< status=success
packet:          git< 0000
packet:          git< SMUDGED_CONTENT
packet:          git< 0000
packet:          git< 0000  # empty list, keep "status=success" unchanged!
```

If the result content is empty then the filter is expected to respond with a "success" status and a flush packet to signal the empty content.

```
packet:          git< status=success
packet:          git< 0000
packet:          git< 0000  # empty content!
```

```
packet:          git< 0000  # empty list, keep "status=success" unchanged!
```

In case the filter cannot or does not want to process the content, it is expected to respond with an "error" status.

```
packet:          git< status=error
packet:          git< 0000
```

If the filter experiences an error during processing, then it can send the status "error" after the content was (partially or completely) sent.

```
packet:          git< status=success
packet:          git< 0000
packet:          git< HALF_WRITTEN_ERRONEOUS_CONTENT
packet:          git< 0000
packet:          git< status=error
packet:          git< 0000
```

In case the filter cannot or does not want to process the content as well as any future content for the lifetime of the Git process, then it is expected to respond with an "abort" status at any point in the protocol.

```
packet:          git< status=abort
packet:          git< 0000
```

Git neither stops nor restarts the filter process in case the "error"/"abort" status is set. However, Git sets its exit code according to the *filter.<driver>.required* flag, mimicking the behavior of the *filter.<driver>.clean* / *filter.<driver>.smudge* mechanism.

If the filter dies during the communication or does not adhere to the protocol then Git will stop the filter process and restart it with the next file that needs to be processed. Depending on the *filter.<driver>.required* flag Git will interpret that as error.

1.9. Delay

If the filter supports the "delay" capability, then Git can send the flag "can-delay" after the filter command and pathname. This flag denotes that the filter can delay filtering the current blob (e.g. to compensate network latencies) by responding with no content but with the status "delayed" and a flush packet.

```
packet:          git> command=smudge
packet:          git> pathname=path/testfile.dat
packet:          git> can-delay=1
packet:          git> 0000
packet:          git> CONTENT
packet:          git> 0000
packet:          git< status=delayed
packet:          git< 0000
```

If the filter supports the "delay" capability then it must support the "list_available_blobs" command. If Git sends this command, then the filter is expected to return a list of pathnames representing blobs that have been delayed earlier and are now available. The list must be terminated with a flush packet followed by a "success" status that is also terminated with a flush packet. If no blobs for the delayed paths are available, yet, then the filter is expected to block the response until at least one blob becomes available. The filter can tell Git that it has no more delayed blobs by sending an empty list. As soon as the filter responds with an empty list, Git stops asking. All blobs that Git has not received at this point are considered missing and will result in an error.

```
packet:          git> command=list_available_blobs
packet:          git> 0000
packet:          git< pathname=path/testfile.dat
packet:          git< pathname=path/otherfile.dat
packet:          git< 0000
packet:          git< status=success
packet:          git< 0000
```

After Git received the pathnames, it will request the corresponding blobs again. These requests contain a pathname and an empty content section. The filter is expected to respond with the smudged content in the usual way as explained above.

```
packet:          git> command=smudge
packet:          git> pathname=path/testfile.dat
packet:          git> 0000
packet:          git> 0000  # empty content!
packet:          git< status=success
packet:          git< 0000
packet:          git< SMUDGED_CONTENT
packet:          git< 0000
packet:          git< 0000  # empty list, keep "status=success" unchanged!
```

1.10. Example

A long running filter demo implementation can be found in *contrib/long-running-filter/example.pl* located in the Git core repository. If you develop your own long running filter process then the *GIT_TRACE_PACKET* environment variables can be very helpful for debugging (see Section G.3.1, “git(1)”).

Please note that you cannot use an existing *filter.<driver>.clean* or *filter.<driver>.smudge* command with *filter.<driver>.process* because the former two use a different inter process communication protocol than the latter one.

1.11. Interaction between checkin/checkout attributes

In the check-in codepath, the worktree file is first converted with *filter* driver (if specified and corresponding driver defined), then the result is processed with *ident* (if specified), and then finally with *text* (again, if specified and applicable).

In the check-out codepath, the blob content is first converted with *text*, and then *ident* and fed to *filter*.

1.12. Merging branches with differing checkin/checkout attributes

If you have added attributes to a file that cause the canonical repository format for that file to change, such as adding a clean/smudge filter or text/eol/ident attributes, merging anything where the attribute is not in place would normally cause merge conflicts.

To prevent these unnecessary merge conflicts, Git can be told to run a virtual check-out and check-in of all three stages of a file when resolving a three-way merge by setting the *merge.renormalize* configuration variable. This prevents changes caused by check-in conversion from causing spurious merge conflicts when a converted file is merged with an unconverted file.

As long as a "smudge#clean" results in the same output as a "clean" even on files that are already smudged, this strategy will automatically resolve all filter-related conflicts. Filters that do not act in this way may cause additional merge conflicts that must be resolved manually.

2. Generating diff text

2.1. *diff*

The attribute *diff* affects how Git generates diffs for particular files. It can tell Git whether to generate a textual patch for the path or to treat the path as a binary file. It can also affect what line is shown on the hunk header *@@ -k,l +n,m @@* line, tell Git to use an external command to generate the diff, or ask Git to convert binary files to a text format before generating the diff.

Set

A path to which the *diff* attribute is set is treated as text, even when they contain byte values that normally never appear in text files, such as NUL.

Unset

A path to which the *diff* attribute is unset will generate *Binary files differ* (or a binary patch, if binary patches are enabled).

Unspecified

A path to which the *diff* attribute is unspecified first gets its contents inspected, and if it looks like text and is smaller than core.bigFileThreshold, it is treated as text. Otherwise it would generate *Binary files differ*.

String

Diff is shown using the specified diff driver. Each driver may specify one or more options, as described in the following section. The options for the diff driver "foo" are defined by the configuration variables in the "diff.foo" section of the Git config file.

2.2. Defining an external diff driver

The definition of a diff driver is done in *gitconfig*, not *gitattributes* file, so strictly speaking this manual page is a wrong place to talk about it. However…

To define an external diff driver *jcdiff*, add a section to your *$GIT_DIR/config* file (or *$HOME/.gitconfig* file) like this:

```
[diff "jcdiff"]
        command = j-c-diff
```

When Git needs to show you a diff for the path with *diff* attribute set to *jcdiff*, it calls the command you specified with the above configuration, i.e. *j-c-diff*, with 7 parameters, just like *GIT_EXTERNAL_DIFF* program is called. See Section G.3.1, “git(1)” for details.

2.3. Setting the internal diff algorithm

The diff algorithm can be set through the *diff.algorithm* config key, but sometimes it may be helpful to set the diff algorithm per path. For example, one may want to use the *minimal* diff algorithm for .json files, and the *histogram* for .c files, and so on without having to pass in the algorithm through the command line each time.

First, in *.gitattributes*, assign the *diff* attribute for paths.

```
*.json diff=<name>
```

Then, define a "diff.<name>.algorithm" configuration to specify the diff algorithm, choosing from *myers*, *patience*, *minimal*, or *histogram*.

```
[diff "<name>"]
  algorithm = histogram
```

This diff algorithm applies to user facing diff output like git-diff(1), git-show(1) and is used for the *--stat* output as well. The merge machinery will not use the diff algorithm set through this method.

> **Note**
>
> If *diff.<name>.command* is defined for path with the *diff=<name>* attribute, it is executed as an external diff driver (see above), and adding *diff.<name>.algorithm* has no effect, as the algorithm is not passed to the external diff driver.

2.4. Defining a custom hunk-header

Each group of changes (called a "hunk") in the textual diff output is prefixed with a line of the form:

```
@@ -k,l +n,m @@ TEXT
```

This is called a *hunk header*. The "TEXT" portion is by default a line that begins with an alphabet, an underscore or a dollar sign; this matches what GNU *diff -p* output uses. This default selection however is not suited for some contents, and you can use a customized pattern to make a selection.

First, in .gitattributes, you would assign the *diff* attribute for paths.

```
*.tex   diff=tex
```

Then, you would define a "diff.tex.xfuncname" configuration to specify a regular expression that matches a line that you would want to appear as the hunk header "TEXT". Add a section to your *$GIT_DIR/config* file (or *$HOME/.gitconfig* file) like this:

```
[diff "tex"]
        xfuncname = "^(\\\\(sub)*section\\{.*)$"
```

Note. A single level of backslashes are eaten by the configuration file parser, so you would need to double the backslashes; the pattern above picks a line that begins with a backslash, and zero or more occurrences of *sub* followed by *section* followed by open brace, to the end of line.

There are a few built-in patterns to make this easier, and *tex* is one of them, so you do not have to write the above in your configuration file (you still need to enable this with the attribute mechanism, via *.gitattributes*). The following built in patterns are available:

- *ada* suitable for source code in the Ada language.
- *bash* suitable for source code in the Bourne-Again SHell language. Covers a superset of POSIX shell function definitions.
- *bibtex* suitable for files with BibTeX coded references.
- *cpp* suitable for source code in the C and C++ languages.
- *csharp* suitable for source code in the C# language.
- *css* suitable for cascading style sheets.
- *dts* suitable for devicetree (DTS) files.
- *elixir* suitable for source code in the Elixir language.
- *fortran* suitable for source code in the Fortran language.
- *fountain* suitable for Fountain documents.
- *golang* suitable for source code in the Go language.
- *html* suitable for HTML/XHTML documents.
- *java* suitable for source code in the Java language.
- *kotlin* suitable for source code in the Kotlin language.
- *markdown* suitable for Markdown documents.
- *matlab* suitable for source code in the MATLAB and Octave languages.
- *objc* suitable for source code in the Objective-C language.
- *pascal* suitable for source code in the Pascal/Delphi language.

- *perl* suitable for source code in the Perl language.
- *php* suitable for source code in the PHP language.
- *python* suitable for source code in the Python language.
- *ruby* suitable for source code in the Ruby language.
- *rust* suitable for source code in the Rust language.
- *scheme* suitable for source code in the Scheme language.
- *tex* suitable for source code for LaTeX documents.

2.5. Customizing word diff

You can customize the rules that *git diff --word-diff* uses to split words in a line, by specifying an appropriate regular expression in the "diff.*.wordRegex" configuration variable. For example, in TeX a backslash followed by a sequence of letters forms a command, but several such commands can be run together without intervening whitespace. To separate them, use a regular expression in your *$GIT_DIR/config* file (or *$HOME/.gitconfig* file) like this:

```
[diff "tex"]
        wordRegex = "\\\\[a-zA-Z]+|[{}]|\\\\.|[^\\{}[:space:]]+"
```

A built-in pattern is provided for all languages listed in the previous section.

2.6. Performing text diffs of binary files

Sometimes it is desirable to see the diff of a text-converted version of some binary files. For example, a word processor document can be converted to an ASCII text representation, and the diff of the text shown. Even though this conversion loses some information, the resulting diff is useful for human viewing (but cannot be applied directly).

The *textconv* config option is used to define a program for performing such a conversion. The program should take a single argument, the name of a file to convert, and produce the resulting text on stdout.

For example, to show the diff of the exif information of a file instead of the binary information (assuming you have the exif tool installed), add the following section to your *$GIT_DIR/config* file (or *$HOME/.gitconfig* file):

```
[diff "jpg"]
        textconv = exif
```

> **Note**
>
> The text conversion is generally a one-way conversion; in this example, we lose the actual image contents and focus just on the text data. This means that diffs generated by textconv are *not* suitable for applying. For this reason, only *git diff* and the *git log* family of commands (i.e., log, whatchanged, show) will perform text conversion. *git format-patch* will never generate this output. If you want to send somebody a text-converted diff of a binary file (e.g., because it quickly conveys the changes you have made), you should generate it separately and send it as a comment *in addition to* the usual binary diff that you might send.

Because text conversion can be slow, especially when doing a large number of them with *git log -p*, Git provides a mechanism to cache the output and use it in future diffs. To enable caching, set the "cachetextconv" variable in your diff driver's config. For example:

```
[diff "jpg"]
        textconv = exif
        cachetextconv = true
```

This will cache the result of running "exif" on each blob indefinitely. If you change the textconv config variable for a diff driver, Git will automatically invalidate the cache entries and re-run the textconv filter. If you want to invalidate the cache manually (e.g., because your version of "exif" was updated and now produces better output), you can remove the cache manually with *git update-ref -d refs/notes/textconv/jpg* (where "jpg" is the name of the diff driver, as in the example above).

2.7. Choosing textconv versus external diff

If you want to show differences between binary or specially-formatted blobs in your repository, you can choose to use either an external diff command, or to use textconv to convert them to a diff-able text format. Which method you choose depends on your exact situation.

The advantage of using an external diff command is flexibility. You are not bound to find line-oriented changes, nor is it necessary for the output to resemble unified diff. You are free to locate and report changes in the most appropriate way for your data format.

A textconv, by comparison, is much more limiting. You provide a transformation of the data into a line-oriented text format, and Git uses its regular diff tools to generate the output. There are several advantages to choosing this method:

1. Ease of use. It is often much simpler to write a binary to text transformation than it is to perform your own diff. In many cases, existing programs can be used as textconv filters (e.g., exif, odt2txt).

2. Git diff features. By performing only the transformation step yourself, you can still utilize many of Git's diff features, including colorization, word-diff, and combined diffs for merges.

3. Caching. Textconv caching can speed up repeated diffs, such as those you might trigger by running *git log -p*.

2.8. Marking files as binary

Git usually guesses correctly whether a blob contains text or binary data by examining the beginning of the contents. However, sometimes you may want to override its decision, either because a blob contains binary data later in the file, or because the content, while technically composed of text characters, is opaque to a human reader. For example, many postscript files contain only ASCII characters, but produce noisy and meaningless diffs.

The simplest way to mark a file as binary is to unset the diff attribute in the *.gitattributes* file:

```
*.ps -diff
```

This will cause Git to generate *Binary files differ* (or a binary patch, if binary patches are enabled) instead of a regular diff.

However, one may also want to specify other diff driver attributes. For example, you might want to use *textconv* to convert postscript files to an ASCII representation for human viewing, but otherwise treat them as binary files. You cannot specify both *-diff* and *diff=ps* attributes. The solution is to use the *diff.*.binary* config option:

```
[diff "ps"]
  textconv = ps2ascii
  binary = true
```

3. Performing a three-way merge

3.1. *merge*

The attribute *merge* affects how three versions of a file are merged when a file-level merge is necessary during *git merge*, and other commands such as *git revert* and *git cherry-pick*.

Set

Built-in 3-way merge driver is used to merge the contents in a way similar to *merge* command of *RCS* suite. This is suitable for ordinary text files.

Unset

Take the version from the current branch as the tentative merge result, and declare that the merge has conflicts. This is suitable for binary files that do not have a well-defined merge semantics.

Unspecified

By default, this uses the same built-in 3-way merge driver as is the case when the *merge* attribute is set. However, the *merge.default* configuration variable can name different merge driver to be used with paths for which the *merge* attribute is unspecified.

String

3-way merge is performed using the specified custom merge driver. The built-in 3-way merge driver can be explicitly specified by asking for "text" driver; the built-in "take the current branch" driver can be requested with "binary".

3.2. Built-in merge drivers

There are a few built-in low-level merge drivers defined that can be asked for via the *merge* attribute.

text

Usual 3-way file level merge for text files. Conflicted regions are marked with conflict markers `<<<<<<<`, `=======` and `>>>>>>>`. The version from your branch appears before the `=======` marker, and the version from the merged branch appears after the `=======` marker.

binary

Keep the version from your branch in the work tree, but leave the path in the conflicted state for the user to sort out.

union

Run 3-way file level merge for text files, but take lines from both versions, instead of leaving conflict markers. This tends to leave the added lines in the resulting file in random order and the user should verify the result. Do not use this if you do not understand the implications.

3.3. Defining a custom merge driver

The definition of a merge driver is done in the *.git/config* file, not in the *gitattributes* file, so strictly speaking this manual page is a wrong place to talk about it. However…

To define a custom merge driver *filfre*, add a section to your *$GIT_DIR/config* file (or *$HOME/.gitconfig* file) like this:

```
[merge "filfre"]
        name = feel-free merge driver
        driver = filfre %O %A %B %L %P
        recursive = binary
```

The *merge.*.name* variable gives the driver a human-readable name.

The merge.*.driver` variable's value is used to construct a command to run to common ancestor's version (*%O*), current version (*%A*) and the other branches version (*%B*). These three tokens are replaced with the names of temporary files that hold the contents of these versions when the command line is built. Additionally, *%L* will be replaced with the conflict marker size (see below).

The merge driver is expected to leave the result of the merge in the file named with *%A* by overwriting it, and exit with zero status if it managed to merge them cleanly, or non-zero if there were conflicts. When the driver crashes

(e.g. killed by SEGV), it is expected to exit with non-zero status that are higher than 128, and in such a case, the merge results in a failure (which is different from producing a conflict).

The *merge.*.recursive* variable specifies what other merge driver to use when the merge driver is called for an internal merge between common ancestors, when there are more than one. When left unspecified, the driver itself is used for both internal merge and the final merge.

The merge driver can learn the pathname in which the merged result will be stored via placeholder *%P*. The conflict labels to be used for the common ancestor, local head and other head can be passed by using *%S*, *%X* and '%Y` respectively.

3.4. *conflict-marker-size*

This attribute controls the length of conflict markers left in the work tree file during a conflicted merge. Only a positive integer has a meaningful effect.

For example, this line in *.gitattributes* can be used to tell the merge machinery to leave much longer (instead of the usual 7-character-long) conflict markers when merging the file *Documentation/git-merge.txt* results in a conflict.

```
Documentation/git-merge.txt      conflict-marker-size=32
```

4. Checking whitespace errors

4.1. *whitespace*

The *core.whitespace* configuration variable allows you to define what *diff* and *apply* should consider whitespace errors for all paths in the project (See Section G.3.29, “git-config(1)”). This attribute gives you finer control per path.

Set

Notice all types of potential whitespace errors known to Git. The tab width is taken from the value of the *core.whitespace* configuration variable.

Unset

Do not notice anything as error.

Unspecified

Use the value of the *core.whitespace* configuration variable to decide what to notice as error.

String

Specify a comma separated list of common whitespace problems to notice in the same format as the *core.whitespace* configuration variable.

5. Creating an archive

5.1. *export-ignore*

Files and directories with the attribute *export-ignore* won't be added to archive files.

5.2. *export-subst*

If the attribute *export-subst* is set for a file then Git will expand several placeholders when adding this file to an archive. The expansion depends on the availability of a commit ID, i.e., if Section G.3.7, “git-archive(1)” has been given a tree instead of a commit or a tag then no replacement will be done. The placeholders are the same as those for the option *--pretty=format:* of Section G.3.74, “git-log(1)”, except that they need to be wrapped like this: *$Format:PLACEHOLDERS$* in the file. E.g. the string *$Format:%H$* will be replaced by the commit hash. However, only one *%(describe)* placeholder is expanded per archive to avoid denial-of-service attacks.

6. Packing objects

6.1. *delta*

Delta compression will not be attempted for blobs for paths with the attribute *delta* set to false.

7. Viewing files in GUI tools

7.1. *encoding*

The value of this attribute specifies the character encoding that should be used by GUI tools (e.g. Section G.4.8, “gitk(1)” and Section G.3.61, “git-gui(1)”) to display the contents of the relevant file. Note that due to performance considerations Section G.4.8, “gitk(1)” does not use this attribute unless you manually enable per-file encodings in its options.

If this attribute is not set or has an invalid value, the value of the *gui.encoding* configuration variable is used instead (See Section G.3.29, “git-config(1)”).

USING MACRO ATTRIBUTES

You do not want any end-of-line conversions applied to, nor textual diffs produced for, any binary file you track. You would need to specify e.g.

```
*.jpg -text -diff
```

but that may become cumbersome, when you have many attributes. Using macro attributes, you can define an attribute that, when set, also sets or unsets a number of other attributes at the same time. The system knows a built-in macro attribute, *binary*:

```
*.jpg binary
```

Setting the "binary" attribute also unsets the "text" and "diff" attributes as above. Note that macro attributes can only be "Set", though setting one might have the effect of setting or unsetting other attributes or even returning other attributes to the "Unspecified" state.

DEFINING MACRO ATTRIBUTES

Custom macro attributes can be defined only in top-level gitattributes files (*$GIT_DIR/info/attributes*, the *.gitattributes* file at the top level of the working tree, or the global or system-wide gitattributes files), not in *.gitattributes* files in working tree subdirectories. The built-in macro attribute "binary" is equivalent to:

```
[attr]binary -diff -merge -text
```

NOTES

Git does not follow symbolic links when accessing a *.gitattributes* file in the working tree. This keeps behavior consistent when the file is accessed from the index or a tree versus from the filesystem.

EXAMPLES

If you have these three *gitattributes* file:

```
(in $GIT_DIR/info/attributes)

a*      foo !bar -baz

(in .gitattributes)
abc     foo bar baz

(in t/.gitattributes)
```

```
ab*     merge=filfre
abc     -foo -bar
*.c     frotz
```

the attributes given to path *t/abc* are computed as follows:

1. By examining *t/.gitattributes* (which is in the same directory as the path in question), Git finds that the first line matches. *merge* attribute is set. It also finds that the second line matches, and attributes *foo* and *bar* are unset.

2. Then it examines *.gitattributes* (which is in the parent directory), and finds that the first line matches, but *t/.gitattributes* file already decided how *merge*, *foo* and *bar* attributes should be given to this path, so it leaves *foo* and *bar* unset. Attribute *baz* is set.

3. Finally it examines *$GIT_DIR/info/attributes*. This file is used to override the in-tree settings. The first line is a match, and *foo* is set, *bar* is reverted to unspecified state, and *baz* is unset.

As the result, the attributes assignment to *t/abc* becomes:

```
foo     set to true
bar     unspecified
baz     set to false
merge   set to string value "filfre"
frotz   unspecified
```

SEE ALSO

Section G.3.14, “git-check-attr(1)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.3. gitcredentials(7)

2024-07-02

Revision History

2024-07-02

NAME

gitcredentials - Providing usernames and passwords to Git

SYNOPSIS

```
git config credential.https://example.com.username myusername
git config credential.helper "$helper $options"
```

DESCRIPTION

Git will sometimes need credentials from the user in order to perform operations; for example, it may need to ask for a username and password in order to access a remote repository over HTTP. Some remotes accept a personal access token or OAuth access token as a password. This manual describes the mechanisms Git uses to request these credentials, as well as some features to avoid inputting these credentials repeatedly.

REQUESTING CREDENTIALS

Without any credential helpers defined, Git will try the following strategies to ask the user for usernames and passwords:

1. If the *GIT_ASKPASS* environment variable is set, the program specified by the variable is invoked. A suitable prompt is provided to the program on the command line, and the user's input is read from its standard output.

2. Otherwise, if the *core.askPass* configuration variable is set, its value is used as above.

3. Otherwise, if the *SSH_ASKPASS* environment variable is set, its value is used as above.

4. Otherwise, the user is prompted on the terminal.

AVOIDING REPETITION

It can be cumbersome to input the same credentials over and over. Git provides two methods to reduce this annoyance:

1. Static configuration of usernames for a given authentication context.

2. Credential helpers to cache or store passwords, or to interact with a system password wallet or keychain.

The first is simple and appropriate if you do not have secure storage available for a password. It is generally configured by adding this to your config:

```
[credential "https://example.com"]
        username = me
```

Credential helpers, on the other hand, are external programs from which Git can request both usernames and passwords; they typically interface with secure storage provided by the OS or other programs. Alternatively, a credential-generating helper might generate credentials for certain servers via some API.

To use a helper, you must first select one to use. Git currently includes the following helpers:

cache

Cache credentials in memory for a short period of time. See Section G.3.33, “git-credential-cache(1)” for details.

store

Store credentials indefinitely on disk. See Section G.3.34, “git-credential-store(1)” for details.

You may also have third-party helpers installed; search for *credential-** in the output of *git help -a*, and consult the documentation of individual helpers. Once you have selected a helper, you can tell Git to use it by putting its name into the credential.helper variable.

1. Find a helper.

   ```
   $ git help -a | grep credential-
   credential-foo
   ```

2. Read its description.

   ```
   $ git help credential-foo
   ```

3. Tell Git to use it.

   ```
   $ git config --global credential.helper foo
   ```

1. Available helpers

The community maintains a comprehensive list of Git credential helpers at https://git-scm.com/doc/credential-helpers.

2. OAuth

An alternative to inputting passwords or personal access tokens is to use an OAuth credential helper. Initial authentication opens a browser window to the host. Subsequent authentication happens in the background. Many popular Git hosts support OAuth.

CREDENTIAL CONTEXTS

Git considers each credential to have a context defined by a URL. This context is used to look up context-specific configuration, and is passed to any helpers, which may use it as an index into secure storage.

For instance, imagine we are accessing *https://example.com/foo.git*. When Git looks into a config file to see if a section matches this context, it will consider the two a match if the context is a more-specific subset of the pattern in the config file. For example, if you have this in your config file:

```
[credential "https://example.com"]
        username = foo
```

then we will match: both protocols are the same, both hosts are the same, and the "pattern" URL does not care about the path component at all. However, this context would not match:

```
[credential "https://kernel.org"]
        username = foo
```

because the hostnames differ. Nor would it match *foo.example.com*; Git compares hostnames exactly, without considering whether two hosts are part of the same domain. Likewise, a config entry for *http://example.com* would not match: Git compares the protocols exactly. However, you may use wildcards in the domain name and other pattern matching techniques as with the *http.<URL>.** options.

If the "pattern" URL does include a path component, then this too must match exactly: the context *https://example.com/bar/baz.git* will match a config entry for *https://example.com/bar/baz.git* (in addition to matching the config entry for *https://example.com*) but will not match a config entry for *https://example.com/bar*.

CONFIGURATION OPTIONS

Options for a credential context can be configured either in *credential.** (which applies to all credentials), or *credential.<URL>.**, where <URL> matches the context as described above.

The following options are available in either location:

helper

The name of an external credential helper, and any associated options. If the helper name is not an absolute path, then the string *git credential-* is prepended. The resulting string is executed by the shell (so, for example, setting this to *foo --option=bar* will execute *git credential-foo --option=bar* via the shell. See the manual of specific helpers for examples of their use.

If there are multiple instances of the *credential.helper* configuration variable, each helper will be tried in turn, and may provide a username, password, or nothing. Once Git has acquired both a username and a non-expired password, no more helpers will be tried.

If *credential.helper* is configured to the empty string, this resets the helper list to empty (so you may override a helper set by a lower-priority config file by configuring the empty-string helper, followed by whatever set of helpers you would like).

username

A default username, if one is not provided in the URL.

useHttpPath

By default, Git does not consider the "path" component of an http URL to be worth matching via external helpers. This means that a credential stored for *https://example.com/foo.git* will also be used for *https://example.com/bar.git*. If you do want to distinguish these cases, set this option to *true*.

CUSTOM HELPERS

You can write your own custom helpers to interface with any system in which you keep credentials.

Credential helpers are programs executed by Git to fetch or save credentials from and to long-term storage (where "long-term" is simply longer than a single Git process; e.g., credentials may be stored in-memory for a few minutes, or indefinitely on disk).

Each helper is specified by a single string in the configuration variable *credential.helper* (and others, see Section G.3.29, “git-config(1)”). The string is transformed by Git into a command to be executed using these rules:

1. If the helper string begins with "!", it is considered a shell snippet, and everything after the "!" becomes the command.

2. Otherwise, if the helper string begins with an absolute path, the verbatim helper string becomes the command.

3. Otherwise, the string "git credential-" is prepended to the helper string, and the result becomes the command.

The resulting command then has an "operation" argument appended to it (see below for details), and the result is executed by the shell.

Here are some example specifications:

```
# run "git credential-foo"
[credential]
        helper = foo

# same as above, but pass an argument to the helper
[credential]
        helper = "foo --bar=baz"

# the arguments are parsed by the shell, so use shell
# quoting if necessary
[credential]
        helper = "foo --bar='whitespace arg'"

# you can also use an absolute path, which will not use the git wrapper
[credential]
        helper = "/path/to/my/helper --with-arguments"

# or you can specify your own shell snippet
[credential "https://example.com"]
        username = your_user
        helper = "!f() { test \"$1\" = get && echo \"password=$(cat
 $HOME/.secret)\"; }; f"
```

Generally speaking, rule (3) above is the simplest for users to specify. Authors of credential helpers should make an effort to assist their users by naming their program "git-credential-$NAME", and putting it in the *$PATH* or *$GIT_EXEC_PATH* during installation, which will allow a user to enable it with *git config credential.helper $NAME*.

When a helper is executed, it will have one "operation" argument appended to its command line, which is one of:

get

Return a matching credential, if any exists.

store

Store the credential, if applicable to the helper.

erase

Remove matching credentials, if any, from the helper's storage.

The details of the credential will be provided on the helper's stdin stream. The exact format is the same as the input/output format of the *git credential* plumbing command (see the section *INPUT/OUTPUT FORMAT* in Section G.3.31, “git-credential(1)” for a detailed specification).

For a *get* operation, the helper should produce a list of attributes on stdout in the same format (see Section G.3.31, “git-credential(1)” for common attributes). A helper is free to produce a subset, or even no values at all if it has nothing useful to provide. Any provided attributes will overwrite those already known about by Git's credential subsystem. Unrecognised attributes are silently discarded.

While it is possible to override all attributes, well behaving helpers should refrain from doing so for any attribute other than username and password.

If a helper outputs a *quit* attribute with a value of *true* or *1*, no further helpers will be consulted, nor will the user be prompted (if no credential has been provided, the operation will then fail).

Similarly, no more helpers will be consulted once both username and password had been provided.

For a *store* or *erase* operation, the helper's output is ignored.

If a helper fails to perform the requested operation or needs to notify the user of a potential issue, it may write to stderr.

If it does not support the requested operation (e.g., a read-only store or generator), it should silently ignore the request.

If a helper receives any other operation, it should silently ignore the request. This leaves room for future operations to be added (older helpers will just ignore the new requests).

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.4. gitdiffcore(7)

2024-07-02

Revision History

2024-07-02

NAME

gitdiffcore - Tweaking diff output

SYNOPSIS

*git diff **

DESCRIPTION

The diff commands *git diff-index*, *git diff-files*, and *git diff-tree* can be told to manipulate differences they find in unconventional ways before showing *diff* output. The manipulation is collectively called "diffcore transformation". This short note describes what they are and how to use them to produce *diff* output that is easier to understand than the conventional kind.

The chain of operation

The *git diff-** family works by first comparing two sets of files:

- *git diff-index* compares contents of a "tree" object and the working directory (when *--cached* flag is not used) or a "tree" object and the index file (when *--cached* flag is used);
- *git diff-files* compares contents of the index file and the working directory;

- *git diff-tree* compares contents of two "tree" objects;

In all of these cases, the commands themselves first optionally limit the two sets of files by any pathspecs given on their command-lines, and compare corresponding paths in the two resulting sets of files.

The pathspecs are used to limit the world diff operates in. They remove the filepairs outside the specified sets of pathnames. E.g. If the input set of filepairs included:

```
:100644 100644 bcd1234... 0123456... M junkfile
```

but the command invocation was *git diff-files myfile*, then the junkfile entry would be removed from the list because only "myfile" is under consideration.

The result of comparison is passed from these commands to what is internally called "diffcore", in a format similar to what is output when the -p option is not used. E.g.

```
in-place edit  :100644 100644 bcd1234... 0123456... M file0
create         :000000 100644 0000000... 1234567... A file4
delete         :100644 000000 1234567... 0000000... D file5
unmerged       :000000 000000 0000000... 0000000... U file6
```

The diffcore mechanism is fed a list of such comparison results (each of which is called "filepair", although at this point each of them talks about a single file), and transforms such a list into another list. There are currently 5 such transformations:

- diffcore-break
- diffcore-rename
- diffcore-merge-broken
- diffcore-pickaxe
- diffcore-order
- diffcore-rotate

These are applied in sequence. The set of filepairs *git diff-** commands find are used as the input to diffcore-break, and the output from diffcore-break is used as the input to the next transformation. The final result is then passed to the output routine and generates either diff-raw format (see Output format sections of the manual for *git diff-** commands) or diff-patch format.

diffcore-break: For Splitting Up Complete Rewrites

The second transformation in the chain is diffcore-break, and is controlled by the -B option to the *git diff-** commands. This is used to detect a filepair that represents "complete rewrite" and break such filepair into two filepairs that represent delete and create. E.g. If the input contained this filepair:

```
:100644 100644 bcd1234... 0123456... M file0
```

and if it detects that the file "file0" is completely rewritten, it changes it to:

```
:100644 000000 bcd1234... 0000000... D file0
:000000 100644 0000000... 0123456... A file0
```

For the purpose of breaking a filepair, diffcore-break examines the extent of changes between the contents of the files before and after modification (i.e. the contents that have "bcd1234…" and "0123456…" as their SHA-1 content ID, in the above example). The amount of deletion of original contents and insertion of new material are added together, and if it exceeds the "break score", the filepair is broken into two. The break score defaults to 50% of the size of the smaller of the original and the result (i.e. if the edit shrinks the file, the size of the result is used;

if the edit lengthens the file, the size of the original is used), and can be customized by giving a number after "-B" option (e.g. "-B75" to tell it to use 75%).

diffcore-rename: For Detecting Renames and Copies

This transformation is used to detect renames and copies, and is controlled by the -M option (to detect renames) and the -C option (to detect copies as well) to the *git diff-** commands. If the input contained these filepairs:

```
:100644 000000 0123456... 0000000... D fileX
:000000 100644 0000000... 0123456... A file0
```

and the contents of the deleted file fileX is similar enough to the contents of the created file file0, then rename detection merges these filepairs and creates:

```
:100644 100644 0123456... 0123456... R100 fileX file0
```

When the "-C" option is used, the original contents of modified files, and deleted files (and also unmodified files, if the "--find-copies-harder" option is used) are considered as candidates of the source files in rename/copy operation. If the input were like these filepairs, that talk about a modified file fileY and a newly created file file0:

```
:100644 100644 0123456... 1234567... M fileY
:000000 100644 0000000... bcd3456... A file0
```

the original contents of fileY and the resulting contents of file0 are compared, and if they are similar enough, they are changed to:

```
:100644 100644 0123456... 1234567... M fileY
:100644 100644 0123456... bcd3456... C100 fileY file0
```

In both rename and copy detection, the same "extent of changes" algorithm used in diffcore-break is used to determine if two files are "similar enough", and can be customized to use a similarity score different from the default of 50% by giving a number after the "-M" or "-C" option (e.g. "-M8" to tell it to use 8/10 = 80%).

Note that when rename detection is on but both copy and break detection are off, rename detection adds a preliminary step that first checks if files are moved across directories while keeping their filename the same. If there is a file added to a directory whose contents are sufficiently similar to a file with the same name that got deleted from a different directory, it will mark them as renames and exclude them from the later quadratic step (the one that pairwise compares all unmatched files to find the "best" matches, determined by the highest content similarity). So, for example, if a deleted docs/ext.txt and an added docs/config/ext.txt are similar enough, they will be marked as a rename and prevent an added docs/ext.md that may be even more similar to the deleted docs/ext.txt from being considered as the rename destination in the later step. For this reason, the preliminary "match same filename" step uses a bit higher threshold to mark a file pair as a rename and stop considering other candidates for better matches. At most, one comparison is done per file in this preliminary pass; so if there are several remaining ext.txt files throughout the directory hierarchy after exact rename detection, this preliminary step may be skipped for those files.

Note. When the "-C" option is used with *--find-copies-harder* option, *git diff-** commands feed unmodified filepairs to diffcore mechanism as well as modified ones. This lets the copy detector consider unmodified files as copy source candidates at the expense of making it slower. Without *--find-copies-harder*, *git diff-** commands can detect copies only if the file that was copied happened to have been modified in the same changeset.

diffcore-merge-broken: For Putting Complete Rewrites Back Together

This transformation is used to merge filepairs broken by diffcore-break, and not transformed into rename/copy by diffcore-rename, back into a single modification. This always runs when diffcore-break is used.

For the purpose of merging broken filepairs back, it uses a different "extent of changes" computation from the ones used by diffcore-break and diffcore-rename. It counts only the deletion from the original, and does not count insertion. If you removed only 10 lines from a 100-line document, even if you added 910 new lines to make a new 1000-line document, you did not do a complete rewrite. diffcore-break breaks such a case in order to help diffcore-rename to consider such filepairs as a candidate of rename/copy detection, but if filepairs broken that

way were not matched with other filepairs to create rename/copy, then this transformation merges them back into the original "modification".

The "extent of changes" parameter can be tweaked from the default 80% (that is, unless more than 80% of the original material is deleted, the broken pairs are merged back into a single modification) by giving a second number to -B option, like these:

- -B50/60 (give 50% "break score" to diffcore-break, use 60% for diffcore-merge-broken).
- -B/60 (the same as above, since diffcore-break defaults to 50%).

Note that earlier implementation left a broken pair as separate creation and deletion patches. This was an unnecessary hack, and the latest implementation always merges all the broken pairs back into modifications, but the resulting patch output is formatted differently for easier review in case of such a complete rewrite by showing the entire contents of the old version prefixed with -, followed by the entire contents of the new version prefixed with +.

diffcore-pickaxe: For Detecting Addition/Deletion of Specified String

This transformation limits the set of filepairs to those that change specified strings between the preimage and the postimage in a certain way. -S<block-of-text> and -G<regular-expression> options are used to specify different ways these strings are sought.

"-S<block-of-text>" detects filepairs whose preimage and postimage have different number of occurrences of the specified block of text. By definition, it will not detect in-file moves. Also, when a changeset moves a file wholesale without affecting the interesting string, diffcore-rename kicks in as usual, and *-S* omits the filepair (since the number of occurrences of that string didn't change in that rename-detected filepair). When used with *--pickaxe-regex*, treat the <block-of-text> as an extended POSIX regular expression to match, instead of a literal string.

"-G<regular-expression>" (mnemonic: grep) detects filepairs whose textual diff has an added or a deleted line that matches the given regular expression. This means that it will detect in-file (or what rename-detection considers the same file) moves, which is noise. The implementation runs diff twice and greps, and this can be quite expensive. To speed things up, binary files without textconv filters will be ignored.

When *-S* or *-G* are used without *--pickaxe-all*, only filepairs that match their respective criterion are kept in the output. When *--pickaxe-all* is used, if even one filepair matches their respective criterion in a changeset, the entire changeset is kept. This behavior is designed to make reviewing changes in the context of the whole changeset easier.

diffcore-order: For Sorting the Output Based on Filenames

This is used to reorder the filepairs according to the user's (or project's) taste, and is controlled by the -O option to the *git diff-** commands.

This takes a text file each of whose lines is a shell glob pattern. Filepairs that match a glob pattern on an earlier line in the file are output before ones that match a later line, and filepairs that do not match any glob pattern are output last.

As an example, a typical orderfile for the core Git probably would look like this:

```
README
Makefile
Documentation
*.h
*.c
t
```

diffcore-rotate: For Changing At Which Path Output Starts

This transformation takes one pathname, and rotates the set of filepairs so that the filepair for the given pathname comes first, optionally discarding the paths that come before it. This is used to implement the *--skip-to* and the --

rotate-to options. It is an error when the specified pathname is not in the set of filepairs, but it is not useful to error out when used with "git log" family of commands, because it is unreasonable to expect that a given path would be modified by each and every commit shown by the "git log" command. For this reason, when used with "git log", the filepair that sorts the same as, or the first one that sorts after, the given pathname is where the output starts.

Use of this transformation combined with diffcore-order will produce unexpected results, as the input to this transformation is likely not sorted when diffcore-order is in effect.

SEE ALSO

Section G.3.44, “git-diff(1)”, Section G.3.41, “git-diff-files(1)”, Section G.3.42, “git-diff-index(1)”, Section G.3.43, “git-diff-tree(1)”, Section G.3.54, “git-format-patch(1)”, Section G.3.74, “git-log(1)”, Section G.4.19, “gitglossary(7)”, *The Git User's Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html]

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.5. gitignore(5)

2024-07-02

Revision History

2024-07-02

NAME

gitignore - Specifies intentionally untracked files to ignore

SYNOPSIS

$XDG_CONFIG_HOME/git/ignore, $GIT_DIR/info/exclude, .gitignore

DESCRIPTION

A *gitignore* file specifies intentionally untracked files that Git should ignore. Files already tracked by Git are not affected; see the NOTES below for details.

Each line in a *gitignore* file specifies a pattern. When deciding whether to ignore a path, Git normally checks *gitignore* patterns from multiple sources, with the following order of precedence, from highest to lowest (within one level of precedence, the last matching pattern decides the outcome):

- Patterns read from the command line for those commands that support them.
- Patterns read from a *.gitignore* file in the same directory as the path, or in any parent directory (up to the top-level of the working tree), with patterns in the higher level files being overridden by those in lower level files down to the directory containing the file. These patterns match relative to the location of the *.gitignore* file. A project normally includes such *.gitignore* files in its repository, containing patterns for files generated as part of the project build.
- Patterns read from *$GIT_DIR/info/exclude*.
- Patterns read from the file specified by the configuration variable *core.excludesFile*.

Which file to place a pattern in depends on how the pattern is meant to be used.

- Patterns which should be version-controlled and distributed to other repositories via clone (i.e., files that all developers will want to ignore) should go into a *.gitignore* file.
- Patterns which are specific to a particular repository but which do not need to be shared with other related repositories (e.g., auxiliary files that live inside the repository but are specific to one user's workflow) should go into the *$GIT_DIR/info/exclude* file.

- Patterns which a user wants Git to ignore in all situations (e.g., backup or temporary files generated by the user's editor of choice) generally go into a file specified by *core.excludesFile* in the user's *~/.gitconfig*. Its default value is $XDG_CONFIG_HOME/git/ignore. If $XDG_CONFIG_HOME is either not set or empty, $HOME/.config/git/ignore is used instead.

The underlying Git plumbing tools, such as *git ls-files* and *git read-tree*, read *gitignore* patterns specified by command-line options, or from files specified by command-line options. Higher-level Git tools, such as *git status* and *git add*, use patterns from the sources specified above.

PATTERN FORMAT

- A blank line matches no files, so it can serve as a separator for readability.
- A line starting with # serves as a comment. Put a backslash ("\") in front of the first hash for patterns that begin with a hash.
- Trailing spaces are ignored unless they are quoted with backslash ("\").
- An optional prefix "*!*" which negates the pattern; any matching file excluded by a previous pattern will become included again. It is not possible to re-include a file if a parent directory of that file is excluded. Git doesn't list excluded directories for performance reasons, so any patterns on contained files have no effect, no matter where they are defined. Put a backslash ("\") in front of the first "*!*" for patterns that begin with a literal "*!*", for example, "*\!important!.txt*".
- The slash "/" is used as the directory separator. Separators may occur at the beginning, middle or end of the *.gitignore* search pattern.
- If there is a separator at the beginning or middle (or both) of the pattern, then the pattern is relative to the directory level of the particular *.gitignore* file itself. Otherwise the pattern may also match at any level below the *.gitignore* level.
- If there is a separator at the end of the pattern then the pattern will only match directories, otherwise the pattern can match both files and directories.
- For example, a pattern *doc/frotz/* matches *doc/frotz* directory, but not *a/doc/frotz* directory; however *frotz/* matches *frotz* and *a/frotz* that is a directory (all paths are relative from the *.gitignore* file).
- An asterisk "*" matches anything except a slash. The character "?" matches any one character except "/". The range notation, e.g. *[a-zA-Z]*, can be used to match one of the characters in a range. See fnmatch(3) and the FNM_PATHNAME flag for a more detailed description.

Two consecutive asterisks ("**") in patterns matched against full pathname may have special meaning:

- A leading "**" followed by a slash means match in all directories. For example, "***/foo*" matches file or directory "*foo*" anywhere, the same as pattern "*foo*". "***/foo/bar*" matches file or directory "*bar*" anywhere that is directly under directory "*foo*".
- A trailing "/**" matches everything inside. For example, "*abc/***" matches all files inside directory "*abc*", relative to the location of the *.gitignore* file, with infinite depth.
- A slash followed by two consecutive asterisks then a slash matches zero or more directories. For example, "*a/**/b*" matches "*a/b*", "*a/x/b*", "*a/x/y/b*" and so on.
- Other consecutive asterisks are considered regular asterisks and will match according to the previous rules.

CONFIGURATION

The optional configuration variable *core.excludesFile* indicates a path to a file containing patterns of file names to exclude, similar to *$GIT_DIR/info/exclude*. Patterns in the exclude file are used in addition to those in *$GIT_DIR/info/exclude*.

NOTES

The purpose of gitignore files is to ensure that certain files not tracked by Git remain untracked.

To stop tracking a file that is currently tracked, use *git rm --cached* to remove the file from the index. The filename can then be added to the *.gitignore* file to stop the file from being reintroduced in later commits.

Git does not follow symbolic links when accessing a *.gitignore* file in the working tree. This keeps behavior consistent when the file is accessed from the index or a tree versus from the filesystem.

EXAMPLES

- The pattern *hello.** matches any file or directory whose name begins with *hello.*. If one wants to restrict this only to the directory and not in its subdirectories, one can prepend the pattern with a slash, i.e. */hello.**; the pattern now matches *hello.txt*, *hello.c* but not *a/hello.java*.
- The pattern *foo/* will match a directory *foo* and paths underneath it, but will not match a regular file or a symbolic link *foo* (this is consistent with the way how pathspec works in general in Git)
- The pattern *doc/frotz* and */doc/frotz* have the same effect in any *.gitignore* file. In other words, a leading slash is not relevant if there is already a middle slash in the pattern.
- The pattern *foo/**, matches *foo/test.json* (a regular file), *foo/bar* (a directory), but it does not match *foo/bar/hello.c* (a regular file), as the asterisk in the pattern does not match *bar/hello.c* which has a slash in it.

```
$ git status
[...]
# Untracked files:
[...]
#       Documentation/foo.html
#       Documentation/gitignore.html
#       file.o
#       lib.a
#       src/internal.o
[...]
$ cat .git/info/exclude
# ignore objects and archives, anywhere in the tree.
*.[oa]
$ cat Documentation/.gitignore
# ignore generated html files,
*.html
# except foo.html which is maintained by hand
!foo.html
$ git status
[...]
# Untracked files:
[...]
#       Documentation/foo.html
[...]
```

Another example:

```
$ cat .gitignore
vmlinux*
$ ls arch/foo/kernel/vm*
arch/foo/kernel/vmlinux.lds.S
$ echo '!/vmlinux*' >arch/foo/kernel/.gitignore
```

The second .gitignore prevents Git from ignoring *arch/foo/kernel/vmlinux.lds.S*.

Example to exclude everything except a specific directory *foo/bar* (note the /* - without the slash, the wildcard would also exclude everything within *foo/bar*):

```
$ cat .gitignore
# exclude everything except directory foo/bar
/*
!/foo
/foo/*
!/foo/bar
```

SEE ALSO

Section G.3.123, “git-rm(1)”, Section G.4.13, “gitrepository-layout(5)”, Section G.3.15, “git-check-ignore(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.6. gitfaq(7)

2024-07-02

Revision History

2024-07-02

NAME

gitfaq - Frequently asked questions about using Git

SYNOPSIS

gitfaq

DESCRIPTION

The examples in this FAQ assume a standard POSIX shell, like *bash* or *dash*, and a user, A U Thor, who has the account *author* on the hosting provider *git.example.org*.

Configuration

What should I put in *user.name*?

> You should put your personal name, generally a form using a given name and family name. For example, the current maintainer of Git uses "Junio C Hamano". This will be the name portion that is stored in every commit you make.
>
> This configuration doesn't have any effect on authenticating to remote services; for that, see *credential.username* in Section G.3.29, “git-config(1)”.

What does *http.postBuffer* really do?

> This option changes the size of the buffer that Git uses when pushing data to a remote over HTTP or HTTPS. If the data is larger than this size, libcurl, which handles the HTTP support for Git, will use chunked transfer encoding since it isn't known ahead of time what the size of the pushed data will be.
>
> Leaving this value at the default size is fine unless you know that either the remote server or a proxy in the middle doesn't support HTTP/1.1 (which introduced the chunked transfer encoding) or is known to be broken with chunked data. This is often (erroneously) suggested as a solution for generic push problems, but since almost every server and proxy supports at least HTTP/1.1, raising this value usually doesn't solve most push problems. A server or proxy that didn't correctly support HTTP/1.1 and chunked transfer encoding wouldn't be that useful on the Internet today, since it would break lots of traffic.

Note that increasing this value will increase the memory used on every relevant push that Git does over HTTP or HTTPS, since the entire buffer is allocated regardless of whether or not it is all used. Thus, it's best to leave it at the default unless you are sure you need a different value.

How do I configure a different editor?

If you haven't specified an editor specifically for Git, it will by default use the editor you've configured using the *VISUAL* or *EDITOR* environment variables, or if neither is specified, the system default (which is usually *vi*). Since some people find *vi* difficult to use or prefer a different editor, it may be desirable to change the editor used.

If you want to configure a general editor for most programs which need one, you can edit your shell configuration (e.g., *~/.bashrc* or *~/.zshenv*) to contain a line setting the *EDITOR* or *VISUAL* environment variable to an appropriate value. For example, if you prefer the editor *nano*, then you could write the following:

```
export VISUAL=nano
```

If you want to configure an editor specifically for Git, you can either set the *core.editor* configuration value or the *GIT_EDITOR* environment variable. You can see Section G.3.152, “git-var(1)” for details on the order in which these options are consulted.

Note that in all cases, the editor value will be passed to the shell, so any arguments containing spaces should be appropriately quoted. Additionally, if your editor normally detaches from the terminal when invoked, you should specify it with an argument that makes it not do that, or else Git will not see any changes. An example of a configuration addressing both of these issues on Windows would be the configuration *"C:\Program Files \Vim\gvim.exe" --nofork*, which quotes the filename with spaces and specifies the *--nofork* option to avoid backgrounding the process.

Credentials

How do I specify my credentials when pushing over HTTP?

The easiest way to do this is to use a credential helper via the *credential.helper* configuration. Most systems provide a standard choice to integrate with the system credential manager. For example, Git for Windows provides the *wincred* credential manager, macOS has the *osxkeychain* credential manager, and Unix systems with a standard desktop environment can use the *libsecret* credential manager. All of these store credentials in an encrypted store to keep your passwords or tokens secure.

In addition, you can use the *store* credential manager which stores in a file in your home directory, or the *cache* credential manager, which does not permanently store your credentials, but does prevent you from being prompted for them for a certain period of time.

You can also just enter your password when prompted. While it is possible to place the password (which must be percent-encoded) in the URL, this is not particularly secure and can lead to accidental exposure of credentials, so it is not recommended.

How do I read a password or token from an environment variable?

The *credential.helper* configuration option can also take an arbitrary shell command that produces the credential protocol on standard output. This is useful when passing credentials into a container, for example.

Such a shell command can be specified by starting the option value with an exclamation point. If your password or token were stored in the *GIT_TOKEN*, you could run the following command to set your credential helper:

```
$ git config credential.helper \
        '!f() { echo username=author; echo "password=$GIT_TOKEN"; };f'
```

How do I change the password or token I've saved in my credential manager?

Usually, if the password or token is invalid, Git will erase it and prompt for a new one. However, there are times when this doesn't always happen. To change the password or token, you can erase the existing credentials

and then Git will prompt for new ones. To erase credentials, use a syntax like the following (substituting your username and the hostname):

```
$ echo url=https://author@git.example.org | git credential reject
```

How do I use multiple accounts with the same hosting provider using HTTP?

Usually the easiest way to distinguish between these accounts is to use the username in the URL. For example, if you have the accounts *author* and *committer* on *git.example.org*, you can use the URLs https://author@git.example.org/org1/project1.git and https://committer@git.example.org/org2/project2.git. This way, when you use a credential helper, it will automatically try to look up the correct credentials for your account. If you already have a remote set up, you can change the URL with something like *git remote set-url origin https://author@git.example.org/org1/project1.git* (see Section G.3.112, "git-remote(1)" for details).

How do I use multiple accounts with the same hosting provider using SSH?

With most hosting providers that support SSH, a single key pair uniquely identifies a user. Therefore, to use multiple accounts, it's necessary to create a key pair for each account. If you're using a reasonably modern OpenSSH version, you can create a new key pair with something like *ssh-keygen -t ed25519 -f ~/.ssh/id_committer*. You can then register the public key (in this case, *~/.ssh/id_committer.pub*; note the *.pub*) with the hosting provider.

Most hosting providers use a single SSH account for pushing; that is, all users push to the *git* account (e.g., *git@git.example.org*). If that's the case for your provider, you can set up multiple aliases in SSH to make it clear which key pair to use. For example, you could write something like the following in *~/.ssh/config*, substituting the proper private key file:

```
# This is the account for author on git.example.org.
Host example_author
        HostName git.example.org
        User git
        # This is the key pair registered for author with
 git.example.org.
        IdentityFile ~/.ssh/id_author
        IdentitiesOnly yes
# This is the account for committer on git.example.org.
Host example_committer
        HostName git.example.org
        User git
        # This is the key pair registered for committer with
 git.example.org.
        IdentityFile ~/.ssh/id_committer
        IdentitiesOnly yes
```

Then, you can adjust your push URL to use *git@example_author* or *git@example_committer* instead of *git@example.org* (e.g., *git remote set-url git@example_author:org1/project1.git*).

Common Issues

I've made a mistake in the last commit. How do I change it?

You can make the appropriate change to your working tree, run *git add <file>* or *git rm <file>*, as appropriate, to stage it, and then *git commit --amend*. Your change will be included in the commit, and you'll be prompted to edit the commit message again; if you wish to use the original message verbatim, you can use the *--no-edit* option to *git commit* in addition, or just save and quit when your editor opens.

I've made a change with a bug and it's been included in the main branch. How should I undo it?

The usual way to deal with this is to use *git revert*. This preserves the history that the original change was made and was a valuable contribution, but also introduces a new commit that undoes those changes because

the original had a problem. The commit message of the revert indicates the commit which was reverted and is usually edited to include an explanation as to why the revert was made.

How do I ignore changes to a tracked file?

Git doesn't provide a way to do this. The reason is that if Git needs to overwrite this file, such as during a checkout, it doesn't know whether the changes to the file are precious and should be kept, or whether they are irrelevant and can safely be destroyed. Therefore, it has to take the safe route and always preserve them.

It's tempting to try to use certain features of *git update-index*, namely the assume-unchanged and skip-worktree bits, but these don't work properly for this purpose and shouldn't be used this way.

If your goal is to modify a configuration file, it can often be helpful to have a file checked into the repository which is a template or set of defaults which can then be copied alongside and modified as appropriate. This second, modified file is usually ignored to prevent accidentally committing it.

I asked Git to ignore various files, yet they are still tracked

A *gitignore* file ensures that certain file(s) which are not tracked by Git remain untracked. However, sometimes particular file(s) may have been tracked before adding them into the *.gitignore*, hence they still remain tracked. To untrack and ignore files/patterns, use *git rm --cached <file/pattern>* and add a pattern to *.gitignore* that matches the <file>. See Section G.4.5, “gitignore(5)” for details.

How do I know if I want to do a fetch or a pull?

A fetch stores a copy of the latest changes from the remote repository, without modifying the working tree or current branch. You can then at your leisure inspect, merge, rebase on top of, or ignore the upstream changes. A pull consists of a fetch followed immediately by either a merge or rebase. See Section G.3.102, “git-pull(1)”.

Merging and Rebasing

What kinds of problems can occur when merging long-lived branches with squash merges?

In general, there are a variety of problems that can occur when using squash merges to merge two branches multiple times. These can include seeing extra commits in *git log* output, with a GUI, or when using the ... notation to express a range, as well as the possibility of needing to re-resolve conflicts again and again.

When Git does a normal merge between two branches, it considers exactly three points: the two branches and a third commit, called the *merge base*, which is usually the common ancestor of the commits. The result of the merge is the sum of the changes between the merge base and each head. When you merge two branches with a regular merge commit, this results in a new commit which will end up as a merge base when they're merged again, because there is now a new common ancestor. Git doesn't have to consider changes that occurred before the merge base, so you don't have to re-resolve any conflicts you resolved before.

When you perform a squash merge, a merge commit isn't created; instead, the changes from one side are applied as a regular commit to the other side. This means that the merge base for these branches won't have changed, and so when Git goes to perform its next merge, it considers all of the changes that it considered the last time plus the new changes. That means any conflicts may need to be re-resolved. Similarly, anything using the ... notation in *git diff*, *git log*, or a GUI will result in showing all of the changes since the original merge base.

As a consequence, if you want to merge two long-lived branches repeatedly, it's best to always use a regular merge commit.

If I make a change on two branches but revert it on one, why does the merge of those branches include the change?

By default, when Git does a merge, it uses a strategy called the *ort* strategy, which does a fancy three-way merge. In such a case, when Git performs the merge, it considers exactly three points: the two heads and a third point, called the *merge base*, which is usually the common ancestor of those commits. Git does not consider the history or the individual commits that have happened on those branches at all.

As a result, if both sides have a change and one side has reverted that change, the result is to include the change. This is because the code has changed on one side and there is no net change on the other, and in this scenario, Git adopts the change.

If this is a problem for you, you can do a rebase instead, rebasing the branch with the revert onto the other branch. A rebase in this scenario will revert the change, because a rebase applies each individual commit, including the revert. Note that rebases rewrite history, so you should avoid rebasing published branches unless you're sure you're comfortable with that. See the NOTES section in Section G.3.107, “git-rebase(1)” for more details.

Hooks

How do I use hooks to prevent users from making certain changes?

The only safe place to make these changes is on the remote repository (i.e., the Git server), usually in the *pre-receive* hook or in a continuous integration (CI) system. These are the locations in which policy can be enforced effectively.

It's common to try to use *pre-commit* hooks (or, for commit messages, *commit-msg* hooks) to check these things, which is great if you're working as a solo developer and want the tooling to help you. However, using hooks on a developer machine is not effective as a policy control because a user can bypass these hooks with *--no-verify* without being noticed (among various other ways). Git assumes that the user is in control of their local repositories and doesn't try to prevent this or tattle on the user.

In addition, some advanced users find *pre-commit* hooks to be an impediment to workflows that use temporary commits to stage work in progress or that create fixup commits, so it's better to push these kinds of checks to the server anyway.

Cross-Platform Issues

I'm on Windows and my text files are detected as binary.

Git works best when you store text files as UTF-8. Many programs on Windows support UTF-8, but some do not and only use the little-endian UTF-16 format, which Git detects as binary. If you can't use UTF-8 with your programs, you can specify a working tree encoding that indicates which encoding your files should be checked out with, while still storing these files as UTF-8 in the repository. This allows tools like Section G.3.44, “git-diff(1)” to work as expected, while still allowing your tools to work.

To do so, you can specify a Section G.4.2, “gitattributes(5)” pattern with the *working-tree-encoding* attribute. For example, the following pattern sets all C files to use UTF-16LE-BOM, which is a common encoding on Windows:

```
*.c     working-tree-encoding=UTF-16LE-BOM
```

You will need to run *git add --renormalize* to have this take effect. Note that if you are making these changes on a project that is used across platforms, you'll probably want to make it in a per-user configuration file or in the one in *$GIT_DIR/info/attributes*, since making it in a *.gitattributes* file in the repository will apply to all users of the repository.

See the following entry for information about normalizing line endings as well, and see Section G.4.2, “gitattributes(5)” for more information about attribute files.

I'm on Windows and git diff shows my files as having a *^M* at the end.

By default, Git expects files to be stored with Unix line endings. As such, the carriage return (*^M*) that is part of a Windows line ending is shown because it is considered to be trailing whitespace. Git defaults to showing trailing whitespace only on new lines, not existing ones.

You can store the files in the repository with Unix line endings and convert them automatically to your platform's line endings. To do that, set the configuration option *core.eol* to *native* and see the following entry for information about how to configure files as text or binary.

You can also control this behavior with the *core.whitespace* setting if you don't wish to remove the carriage returns from your line endings.

Why do I have a file that's always modified?

Internally, Git always stores file names as sequences of bytes and doesn't perform any encoding or case folding. However, Windows and macOS by default both perform case folding on file names. As a result, it's possible to end up with multiple files or directories whose names differ only in case. Git can handle this just fine, but the file system can store only one of these files, so when Git reads the other file to see its contents, it looks modified.

It's best to remove one of the files such that you only have one file. You can do this with commands like the following (assuming two files *AFile.txt* and *afile.txt*) on an otherwise clean working tree:

```
$ git rm --cached AFile.txt
$ git commit -m 'Remove files conflicting in case'
$ git checkout .
```

This avoids touching the disk, but removes the additional file. Your project may prefer to adopt a naming convention, such as all-lowercase names, to avoid this problem from occurring again; such a convention can be checked using a *pre-receive* hook or as part of a continuous integration (CI) system.

It is also possible for perpetually modified files to occur on any platform if a smudge or clean filter is in use on your system but a file was previously committed without running the smudge or clean filter. To fix this, run the following on an otherwise clean working tree:

```
$ git add --renormalize .
```

What's the recommended way to store files in Git?

While Git can store and handle any file of any type, there are some settings that work better than others. In general, we recommend that text files be stored in UTF-8 without a byte-order mark (BOM) with LF (Unix-style) endings. We also recommend the use of UTF-8 (again, without BOM) in commit messages. These are the settings that work best across platforms and with tools such as *git diff* and *git merge*.

Additionally, if you have a choice between storage formats that are text based or non-text based, we recommend storing files in the text format and, if necessary, transforming them into the other format. For example, a text-based SQL dump with one record per line will work much better for diffing and merging than an actual database file. Similarly, text-based formats such as Markdown and AsciiDoc will work better than binary formats such as Microsoft Word and PDF.

Similarly, storing binary dependencies (e.g., shared libraries or JAR files) or build products in the repository is generally not recommended. Dependencies and build products are best stored on an artifact or package server with only references, URLs, and hashes stored in the repository.

We also recommend setting a Section G.4.2, “gitattributes(5)” file to explicitly mark which files are text and which are binary. If you want Git to guess, you can set the attribute *text=auto*. For example, the following might be appropriate in some projects:

```
# By default, guess.
*       text=auto
# Mark all C files as text.
*.c     text
# Mark all JPEG files as binary.
*.jpg   binary
```

These settings help tools pick the right format for output such as patches and result in files being checked out in the appropriate line ending for the platform.

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.7. githooks(5)

2024-07-02

Revision History

2024-07-02

NAME

githooks - Hooks used by Git

SYNOPSIS

$GIT_DIR/hooks/* (or `git config core.hooksPath`/*)

DESCRIPTION

Hooks are programs you can place in a hooks directory to trigger actions at certain points in git's execution. Hooks that don't have the executable bit set are ignored.

By default the hooks directory is *$GIT_DIR/hooks*, but that can be changed via the *core.hooksPath* configuration variable (see Section G.3.29, “git-config(1)”).

Before Git invokes a hook, it changes its working directory to either $GIT_DIR in a bare repository or the root of the working tree in a non-bare repository. An exception are hooks triggered during a push (*pre-receive*, *update*, *post-receive*, *post-update*, *push-to-checkout*) which are always executed in $GIT_DIR.

Environment variables, such as *GIT_DIR*, *GIT_WORK_TREE*, etc., are exported so that Git commands run by the hook can correctly locate the repository. If your hook needs to invoke Git commands in a foreign repository or in a different working tree of the same repository, then it should clear these environment variables so they do not interfere with Git operations at the foreign location. For example:

```
local_desc=$(git describe)
foreign_desc=$(unset $(git rev-parse --local-env-vars); git -C ../foreign-
repo describe)
```

Hooks can get their arguments via the environment, command-line arguments, and stdin. See the documentation for each hook below for details.

git init may copy hooks to the new repository, depending on its configuration. See the "TEMPLATE DIRECTORY" section in Section G.3.71, “git-init(1)” for details. When the rest of this document refers to "default hooks" it's talking about the default template shipped with Git.

The currently supported hooks are described below.

HOOKS

1. applypatch-msg

This hook is invoked by Section G.3.3, “git-am(1)”. It takes a single parameter, the name of the file that holds the proposed commit log message. Exiting with a non-zero status causes *git am* to abort before applying the patch.

The hook is allowed to edit the message file in place, and can be used to normalize the message into some project standard format. It can also be used to refuse the commit after inspecting the message file.

The default *applypatch-msg* hook, when enabled, runs the *commit-msg* hook, if the latter is enabled.

2. pre-applypatch

This hook is invoked by Section G.3.3, “git-am(1)”. It takes no parameter, and is invoked after the patch is applied, but before a commit is made.

If it exits with non-zero status, then the working tree will not be committed after applying the patch.

It can be used to inspect the current working tree and refuse to make a commit if it does not pass certain tests.

The default *pre-applypatch* hook, when enabled, runs the *pre-commit* hook, if the latter is enabled.

3. post-applypatch

This hook is invoked by Section G.3.3, "git-am(1)". It takes no parameter, and is invoked after the patch is applied and a commit is made.

This hook is meant primarily for notification, and cannot affect the outcome of *git am*.

4. pre-commit

This hook is invoked by Section G.3.28, "git-commit(1)", and can be bypassed with the *--no-verify* option. It takes no parameters, and is invoked before obtaining the proposed commit log message and making a commit. Exiting with a non-zero status from this script causes the *git commit* command to abort before creating a commit.

The default *pre-commit* hook, when enabled, catches introduction of lines with trailing whitespaces and aborts the commit when such a line is found.

All the *git commit* hooks are invoked with the environment variable *GIT_EDITOR=:* if the command will not bring up an editor to modify the commit message.

The default *pre-commit* hook, when enabled--and with the *hooks.allownonascii* config option unset or set to false--prevents the use of non-ASCII filenames.

5. pre-merge-commit

This hook is invoked by Section G.3.86, "git-merge(1)", and can be bypassed with the *--no-verify* option. It takes no parameters, and is invoked after the merge has been carried out successfully and before obtaining the proposed commit log message to make a commit. Exiting with a non-zero status from this script causes the *git merge* command to abort before creating a commit.

The default *pre-merge-commit* hook, when enabled, runs the *pre-commit* hook, if the latter is enabled.

This hook is invoked with the environment variable *GIT_EDITOR=:* if the command will not bring up an editor to modify the commit message.

If the merge cannot be carried out automatically, the conflicts need to be resolved and the result committed separately (see Section G.3.86, "git-merge(1)"). At that point, this hook will not be executed, but the *pre-commit* hook will, if it is enabled.

6. prepare-commit-msg

This hook is invoked by Section G.3.28, "git-commit(1)" right after preparing the default log message, and before the editor is started.

It takes one to three parameters. The first is the name of the file that contains the commit log message. The second is the source of the commit message, and can be: *message* (if a *-m* or *-F* option was given); *template* (if a *-t* option was given or the configuration option *commit.template* is set); *merge* (if the commit is a merge or a *.git/MERGE_MSG* file exists); *squash* (if a *.git/SQUASH_MSG* file exists); or *commit*, followed by a commit object name (if a *-c*, *-C* or *--amend* option was given).

If the exit status is non-zero, *git commit* will abort.

The purpose of the hook is to edit the message file in place, and it is not suppressed by the *--no-verify* option. A non-zero exit means a failure of the hook and aborts the commit. It should not be used as a replacement for the pre-commit hook.

The sample *prepare-commit-msg* hook that comes with Git removes the help message found in the commented portion of the commit template.

7. commit-msg

This hook is invoked by Section G.3.28, “git-commit(1)” and Section G.3.86, “git-merge(1)”, and can be bypassed with the *--no-verify* option. It takes a single parameter, the name of the file that holds the proposed commit log message. Exiting with a non-zero status causes the command to abort.

The hook is allowed to edit the message file in place, and can be used to normalize the message into some project standard format. It can also be used to refuse the commit after inspecting the message file.

The default *commit-msg* hook, when enabled, detects duplicate *Signed-off-by* trailers, and aborts the commit if one is found.

8. post-commit

This hook is invoked by Section G.3.28, “git-commit(1)”. It takes no parameters, and is invoked after a commit is made.

This hook is meant primarily for notification, and cannot affect the outcome of *git commit*.

9. pre-rebase

This hook is called by Section G.3.107, “git-rebase(1)” and can be used to prevent a branch from getting rebased. The hook may be called with one or two parameters. The first parameter is the upstream from which the series was forked. The second parameter is the branch being rebased, and is not set when rebasing the current branch.

10. post-checkout

This hook is invoked when a Section G.3.19, “git-checkout(1)” or Section G.3.140, “git-switch(1)” is run after having updated the worktree. The hook is given three parameters: the ref of the previous HEAD, the ref of the new HEAD (which may or may not have changed), and a flag indicating whether the checkout was a branch checkout (changing branches, flag=1) or a file checkout (retrieving a file from the index, flag=0). This hook cannot affect the outcome of *git switch* or *git checkout*, other than that the hook's exit status becomes the exit status of these two commands.

It is also run after Section G.3.24, “git-clone(1)”, unless the *--no-checkout* (*-n*) option is used. The first parameter given to the hook is the null-ref, the second the ref of the new HEAD and the flag is always 1. Likewise for *git worktree add* unless *--no-checkout* is used.

This hook can be used to perform repository validity checks, auto-display differences from the previous HEAD if different, or set working dir metadata properties.

11. post-merge

This hook is invoked by Section G.3.86, “git-merge(1)”, which happens when a *git pull* is done on a local repository. The hook takes a single parameter, a status flag specifying whether or not the merge being done was a squash merge. This hook cannot affect the outcome of *git merge* and is not executed, if the merge failed due to conflicts.

This hook can be used in conjunction with a corresponding pre-commit hook to save and restore any form of metadata associated with the working tree (e.g.: permissions/ownership, ACLS, etc). See contrib/hooks/setgit-perms.perl for an example of how to do this.

12. pre-push

This hook is called by Section G.3.103, “git-push(1)” and can be used to prevent a push from taking place. The hook is called with two parameters which provide the name and location of the destination remote, if a named remote is not being used both values will be the same.

Information about what is to be pushed is provided on the hook's standard input with lines of the form:

```
<local-ref> SP <local-object-name> SP <remote-ref> SP <remote-object-name>
 LF
```

For instance, if the command `git push origin master:foreign` were run the hook would receive a line like the following:

```
refs/heads/master 67890 refs/heads/foreign 12345
```

although the full object name would be supplied. If the foreign ref does not yet exist the *<remote-object-name>* will be the all-zeroes object name. If a ref is to be deleted, the *<local-ref>* will be supplied as *(delete)* and the *<local-object-name>* will be the all-zeroes object name. If the local commit was specified by something other than a name which could be expanded (such as *HEAD~*, or an object name) it will be supplied as it was originally given.

If this hook exits with a non-zero status, *git push* will abort without pushing anything. Information about why the push is rejected may be sent to the user by writing to standard error.

13. pre-receive

This hook is invoked by Section G.3.108, “git-receive-pack(1)” when it reacts to *git push* and updates reference(s) in its repository. Just before starting to update refs on the remote repository, the pre-receive hook is invoked. Its exit status determines the success or failure of the update.

This hook executes once for the receive operation. It takes no arguments, but for each ref to be updated it receives on standard input a line of the format:

```
<old-oid> SP <new-oid> SP <ref-name> LF
```

where *<old-oid>* is the old object name stored in the ref, *<new-oid>* is the new object name to be stored in the ref and *<ref-name>* is the full name of the ref. When creating a new ref, *<old-oid>* is the all-zeroes object name.

If the hook exits with non-zero status, none of the refs will be updated. If the hook exits with zero, updating of individual refs can still be prevented by the *update* hook.

Both standard output and standard error output are forwarded to *git send-pack* on the other end, so you can simply *echo* messages for the user.

The number of push options given on the command line of *git push --push-option=...* can be read from the environment variable *GIT_PUSH_OPTION_COUNT*, and the options themselves are found in *GIT_PUSH_OPTION_0*, *GIT_PUSH_OPTION_1*,… If it is negotiated to not use the push options phase, the environment variables will not be set. If the client selects to use push options, but doesn't transmit any, the count variable will be set to zero, *GIT_PUSH_OPTION_COUNT=0*.

See the section on "Quarantine Environment" in Section G.3.108, “git-receive-pack(1)” for some caveats.

14. update

This hook is invoked by Section G.3.108, “git-receive-pack(1)” when it reacts to *git push* and updates reference(s) in its repository. Just before updating the ref on the remote repository, the update hook is invoked. Its exit status determines the success or failure of the ref update.

The hook executes once for each ref to be updated, and takes three parameters:

- the name of the ref being updated,
- the old object name stored in the ref,
- and the new object name to be stored in the ref.

A zero exit from the update hook allows the ref to be updated. Exiting with a non-zero status prevents *git receive-pack* from updating that ref.

This hook can be used to prevent *forced* update on certain refs by making sure that the object name is a commit object that is a descendant of the commit object named by the old object name. That is, to enforce a "fast-forward only" policy.

It could also be used to log the old..new status. However, it does not know the entire set of branches, so it would end up firing one e-mail per ref when used naively, though. The *post-receive* hook is more suited to that.

In an environment that restricts the users' access only to git commands over the wire, this hook can be used to implement access control without relying on filesystem ownership and group membership. See Section G.3.129, “git-shell(1)” for how you might use the login shell to restrict the user's access to only git commands.

Both standard output and standard error output are forwarded to *git send-pack* on the other end, so you can simply *echo* messages for the user.

The default *update* hook, when enabled--and with *hooks.allowunannotated* config option unset or set to false--prevents unannotated tags from being pushed.

15. proc-receive

This hook is invoked by Section G.3.108, “git-receive-pack(1)”. If the server has set the multi-valued config variable *receive.procReceiveRefs*, and the commands sent to *receive-pack* have matching reference names, these commands will be executed by this hook, instead of by the internal *execute_commands()* function. This hook is responsible for updating the relevant references and reporting the results back to *receive-pack*.

This hook executes once for the receive operation. It takes no arguments, but uses a pkt-line format protocol to communicate with *receive-pack* to read commands, push-options and send results. In the following example for the protocol, the letter *S* stands for *receive-pack* and the letter *H* stands for this hook.

```
# Version and features negotiation.
S: PKT-LINE(version=1\0push-options atomic...)
S: flush-pkt
H: PKT-LINE(version=1\0push-options...)
H: flush-pkt

# Send commands from server to the hook.
S: PKT-LINE(<old-oid> <new-oid> <ref>)
S: ... ...
S: flush-pkt
# Send push-options only if the 'push-options' feature is enabled.
S: PKT-LINE(push-option)
S: ... ...
S: flush-pkt

# Receive results from the hook.
# OK, run this command successfully.
H: PKT-LINE(ok <ref>)
# NO, I reject it.
H: PKT-LINE(ng <ref> <reason>)
# Fall through, let 'receive-pack' execute it.
H: PKT-LINE(ok <ref>)
H: PKT-LINE(option fall-through)
# OK, but has an alternate reference.  The alternate reference name
# and other status can be given in option directives.
H: PKT-LINE(ok <ref>)
H: PKT-LINE(option refname <refname>)
H: PKT-LINE(option old-oid <old-oid>)
H: PKT-LINE(option new-oid <new-oid>)
H: PKT-LINE(option forced-update)
H: ... ...
```

```
H: flush-pkt
```

Each command for the *proc-receive* hook may point to a pseudo-reference and always has a zero-old as its old-oid, while the *proc-receive* hook may update an alternate reference and the alternate reference may exist already with a non-zero old-oid. For this case, this hook will use "option" directives to report extended attributes for the reference given by the leading "ok" directive.

The report of the commands of this hook should have the same order as the input. The exit status of the *proc-receive* hook only determines the success or failure of the group of commands sent to it, unless atomic push is in use.

16. post-receive

This hook is invoked by Section G.3.108, “git-receive-pack(1)” when it reacts to *git push* and updates reference(s) in its repository. It executes on the remote repository once after all the refs have been updated.

This hook executes once for the receive operation. It takes no arguments, but gets the same information as the *pre-receive* hook does on its standard input.

This hook does not affect the outcome of *git receive-pack*, as it is called after the real work is done.

This supersedes the *post-update* hook in that it gets both old and new values of all the refs in addition to their names.

Both standard output and standard error output are forwarded to *git send-pack* on the other end, so you can simply *echo* messages for the user.

The default *post-receive* hook is empty, but there is a sample script *post-receive-email* provided in the *contrib/hooks* directory in Git distribution, which implements sending commit emails.

The number of push options given on the command line of *git push --push-option=...* can be read from the environment variable *GIT_PUSH_OPTION_COUNT*, and the options themselves are found in *GIT_PUSH_OPTION_0*, *GIT_PUSH_OPTION_1*,… If it is negotiated to not use the push options phase, the environment variables will not be set. If the client selects to use push options, but doesn't transmit any, the count variable will be set to zero, *GIT_PUSH_OPTION_COUNT=0*.

17. post-update

This hook is invoked by Section G.3.108, “git-receive-pack(1)” when it reacts to *git push* and updates reference(s) in its repository. It executes on the remote repository once after all the refs have been updated.

It takes a variable number of parameters, each of which is the name of ref that was actually updated.

This hook is meant primarily for notification, and cannot affect the outcome of *git receive-pack*.

The *post-update* hook can tell what are the heads that were pushed, but it does not know what their original and updated values are, so it is a poor place to do log old..new. The *post-receive* hook does get both original and updated values of the refs. You might consider it instead if you need them.

When enabled, the default *post-update* hook runs *git update-server-info* to keep the information used by dumb transports (e.g., HTTP) up to date. If you are publishing a Git repository that is accessible via HTTP, you should probably enable this hook.

Both standard output and standard error output are forwarded to *git send-pack* on the other end, so you can simply *echo* messages for the user.

18. reference-transaction

This hook is invoked by any Git command that performs reference updates. It executes whenever a reference transaction is prepared, committed or aborted and may thus get called multiple times. The hook does not cover symbolic references (but that may change in the future).

The hook takes exactly one argument, which is the current state the given reference transaction is in:

- "prepared": All reference updates have been queued to the transaction and references were locked on disk.
- "committed": The reference transaction was committed and all references now have their respective new value.
- "aborted": The reference transaction was aborted, no changes were performed and the locks have been released.

For each reference update that was added to the transaction, the hook receives on standard input a line of the format:

```
<old-oid> SP <new-oid> SP <ref-name> LF
```

where *<old-oid>* is the old object name passed into the reference transaction, *<new-oid>* is the new object name to be stored in the ref and *<ref-name>* is the full name of the ref. When force updating the reference regardless of its current value or when the reference is to be created anew, *<old-oid>* is the all-zeroes object name. To distinguish these cases, you can inspect the current value of *<ref-name>* via *git rev-parse*.

The exit status of the hook is ignored for any state except for the "prepared" state. In the "prepared" state, a non-zero exit status will cause the transaction to be aborted. The hook will not be called with "aborted" state in that case.

19. push-to-checkout

This hook is invoked by Section G.3.108, “git-receive-pack(1)” when it reacts to *git push* and updates reference(s) in its repository, and when the push tries to update the branch that is currently checked out and the *receive.denyCurrentBranch* configuration variable is set to *updateInstead*. Such a push by default is refused if the working tree and the index of the remote repository has any difference from the currently checked out commit; when both the working tree and the index match the current commit, they are updated to match the newly pushed tip of the branch. This hook is to be used to override the default behaviour.

The hook receives the commit with which the tip of the current branch is going to be updated. It can exit with a non-zero status to refuse the push (when it does so, it must not modify the index or the working tree). Or it can make any necessary changes to the working tree and to the index to bring them to the desired state when the tip of the current branch is updated to the new commit, and exit with a zero status.

For example, the hook can simply run *git read-tree -u -m HEAD "$1"* in order to emulate *git fetch* that is run in the reverse direction with *git push*, as the two-tree form of *git read-tree -u -m* is essentially the same as *git switch* or *git checkout* that switches branches while keeping the local changes in the working tree that do not interfere with the difference between the branches.

20. pre-auto-gc

This hook is invoked by *git gc --auto* (see Section G.3.58, “git-gc(1)”). It takes no parameter, and exiting with non-zero status from this script causes the *git gc --auto* to abort.

21. post-rewrite

This hook is invoked by commands that rewrite commits (Section G.3.28, “git-commit(1)” when called with *--amend* and Section G.3.107, “git-rebase(1)”; however, full-history (re)writing tools like Section G.3.47, “git-fast-import(1)” or git-filter-repo [https://github.com/newren/git-filter-repo] typically do not call it!). Its first argument denotes the command it was invoked by: currently one of *amend* or *rebase*. Further command-dependent arguments may be passed in the future.

The hook receives a list of the rewritten commits on stdin, in the format

```
<old-object-name> SP <new-object-name> [ SP <extra-info> ] LF
```

The *extra-info* is again command-dependent. If it is empty, the preceding SP is also omitted. Currently, no commands pass any *extra-info*.

The hook always runs after the automatic note copying (see "notes.rewrite.<command>" in Section G.3.29, “git-config(1)”) has happened, and thus has access to these notes.

The following command-specific comments apply:

rebase

For the *squash* and *fixup* operation, all commits that were squashed are listed as being rewritten to the squashed commit. This means that there will be several lines sharing the same *new-object-name*.

The commits are guaranteed to be listed in the order that they were processed by rebase.

22. sendemail-validate

This hook is invoked by Section G.3.124, “git-send-email(1)”.

It takes these command line arguments. They are, 1. the name of the file which holds the contents of the email to be sent. 2. The name of the file which holds the SMTP headers of the email.

The SMTP headers are passed in the exact same way as they are passed to the user's Mail Transport Agent (MTA). In effect, the email given to the user's MTA, is the contents of $2 followed by the contents of $1.

An example of a few common headers is shown below. Take notice of the capitalization and multi-line tab structure.

```
From: Example <from@example.com>
To: to@example.com
Cc: cc@example.com,
        A <author@example.com>,
        One <one@example.com>,
        two@example.com
Subject: PATCH-STRING
```

Exiting with a non-zero status causes *git send-email* to abort before sending any e-mails.

The following environment variables are set when executing the hook.

GIT_SENDEMAIL_FILE_COUNTER

A 1-based counter incremented by one for every file holding an e-mail to be sent (excluding any FIFOs). This counter does not follow the patch series counter scheme. It will always start at 1 and will end at GIT_SENDEMAIL_FILE_TOTAL.

GIT_SENDEMAIL_FILE_TOTAL

The total number of files that will be sent (excluding any FIFOs). This counter does not follow the patch series counter scheme. It will always be equal to the number of files being sent, whether there is a cover letter or not.

These variables may for instance be used to validate patch series.

The sample *sendemail-validate* hook that comes with Git checks that all sent patches (excluding the cover letter) can be applied on top of the upstream repository default branch without conflicts. Some placeholders are left for additional validation steps to be performed after all patches of a given series have been applied.

23. fsmonitor-watchman

This hook is invoked when the configuration option *core.fsmonitor* is set to *.git/hooks/fsmonitor-watchman* or *.git/hooks/fsmonitor-watchmanv2* depending on the version of the hook to use.

Version 1 takes two arguments, a version (1) and the time in elapsed nanoseconds since midnight, January 1, 1970.

Version 2 takes two arguments, a version (2) and a token that is used for identifying changes since the token. For watchman this would be a clock id. This version must output to stdout the new token followed by a NUL before the list of files.

The hook should output to stdout the list of all files in the working directory that may have changed since the requested time. The logic should be inclusive so that it does not miss any potential changes. The paths should be relative to the root of the working directory and be separated by a single NUL.

It is OK to include files which have not actually changed. All changes including newly-created and deleted files should be included. When files are renamed, both the old and the new name should be included.

Git will limit what files it checks for changes as well as which directories are checked for untracked files based on the path names given.

An optimized way to tell git "all files have changed" is to return the filename /.

The exit status determines whether git will use the data from the hook to limit its search. On error, it will fall back to verifying all files and folders.

24. p4-changelist

This hook is invoked by *git-p4 submit*.

The *p4-changelist* hook is executed after the changelist message has been edited by the user. It can be bypassed with the *--no-verify* option. It takes a single parameter, the name of the file that holds the proposed changelist text. Exiting with a non-zero status causes the command to abort.

The hook is allowed to edit the changelist file and can be used to normalize the text into some project standard format. It can also be used to refuse the Submit after inspect the message file.

Run *git-p4 submit --help* for details.

25. p4-prepare-changelist

This hook is invoked by *git-p4 submit*.

The *p4-prepare-changelist* hook is executed right after preparing the default changelist message and before the editor is started. It takes one parameter, the name of the file that contains the changelist text. Exiting with a non-zero status from the script will abort the process.

The purpose of the hook is to edit the message file in place, and it is not suppressed by the *--no-verify* option. This hook is called even if *--prepare-p4-only* is set.

Run *git-p4 submit --help* for details.

26. p4-post-changelist

This hook is invoked by *git-p4 submit*.

The *p4-post-changelist* hook is invoked after the submit has successfully occurred in P4. It takes no parameters and is meant primarily for notification and cannot affect the outcome of the git p4 submit action.

Run *git-p4 submit --help* for details.

27. p4-pre-submit

This hook is invoked by *git-p4 submit*. It takes no parameters and nothing from standard input. Exiting with non-zero status from this script prevent *git-p4 submit* from launching. It can be bypassed with the *--no-verify* command line option. Run *git-p4 submit --help* for details.

28. post-index-change

This hook is invoked when the index is written in read-cache.c do_write_locked_index.

The first parameter passed to the hook is the indicator for the working directory being updated. "1" meaning working directory was updated or "0" when the working directory was not updated.

The second parameter passed to the hook is the indicator for whether or not the index was updated and the skip-worktree bit could have changed. "1" meaning skip-worktree bits could have been updated and "0" meaning they were not.

Only one parameter should be set to "1" when the hook runs. The hook running passing "1", "1" should not be possible.

SEE ALSO

Section G.3.64, “git-hook(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.8. gitk(1)

2024-07-02
Revision History
2024-07-02

NAME

gitk - The Git repository browser

SYNOPSIS

gitk [<options>] [<revision-range>] [--] [<path>…]

DESCRIPTION

Displays changes in a repository or a selected set of commits. This includes visualizing the commit graph, showing information related to each commit, and the files in the trees of each revision.

OPTIONS

To control which revisions to show, gitk supports most options applicable to the *git rev-list* command. It also supports a few options applicable to the *git diff-** commands to control how the changes each commit introduces are shown. Finally, it supports some gitk-specific options.

gitk generally only understands options with arguments in the *stuck* form (see Section G.4.1, “gitcli(7)”) due to limitations in the command-line parser.

1. rev-list options and arguments

This manual page describes only the most frequently used options. See Section G.3.120, “git-rev-list(1)” for a complete list.

--all

Show all refs (branches, tags, etc.).

--branches[=<pattern>] , --tags[=<pattern>] , --remotes[=<pattern>]

Pretend as if all the branches (tags, remote branches, resp.) are listed on the command line as *<commit>*. If *<pattern>* is given, limit refs to ones matching given shell glob. If pattern lacks *?*, *, or *[*, */* at the end is implied.

--since=<date>

Show commits more recent than a specific date.

--until=<date>

Show commits older than a specific date.

--date-order

Sort commits by date when possible.

--merge

After an attempt to merge stops with conflicts, show the commits on the history between two branches (i.e. the HEAD and the MERGE_HEAD) that modify the conflicted files and do not exist on all the heads being merged.

--left-right

Mark which side of a symmetric difference a commit is reachable from. Commits from the left side are prefixed with a < symbol and those from the right with a > symbol.

--full-history

When filtering history with <*path*>…, does not prune some history. (See "History simplification" in Section G.3.74, “git-log(1)” for a more detailed explanation.)

--simplify-merges

Additional option to *--full-history* to remove some needless merges from the resulting history, as there are no selected commits contributing to this merge. (See "History simplification" in Section G.3.74, “git-log(1)” for a more detailed explanation.)

--ancestry-path

When given a range of commits to display (e.g. *commit1*..*commit2* or *commit2* ^*commit1*), only display commits that exist directly on the ancestry chain between the *commit1* and *commit2*, i.e. commits that are both descendants of *commit1*, and ancestors of *commit2*. (See "History simplification" in Section G.3.74, “git-log(1)” for a more detailed explanation.)

-L<start>,<end>:<file> , -L:<funcname>:<file>

Trace the evolution of the line range given by <*start*>,<*end*>, or by the function name regex <*funcname*>, within the <*file*>. You may not give any pathspec limiters. This is currently limited to a walk starting from a single revision, i.e., you may only give zero or one positive revision arguments, and <*start*> and <*end*> (or <*funcname*>) must exist in the starting revision. You can specify this option more than once. Implies *--patch*. Patch output can be suppressed using *--no-patch*, but other diff formats (namely *--raw*, *--numstat*, *--shortstat*, *--dirstat*, *--summary*, *--name-only*, *--name-status*, *--check*) are not currently implemented.

<*start*> and <*end*> can take one of these forms:

- number

  If <*start*> or <*end*> is a number, it specifies an absolute line number (lines count from 1).

- */regex/*

  This form will use the first line matching the given POSIX regex. If <*start*> is a regex, it will search from the end of the previous *-L* range, if any, otherwise from the start of file. If <*start*> is *^/regex/*, it will search from the start of file. If <*end*> is a regex, it will search starting at the line given by <*start*>.

- +offset or -offset

  This is only valid for *<end>* and will specify a number of lines before or after the line given by *<start>*.

If *:<funcname>* is given in place of *<start>* and *<end>*, it is a regular expression that denotes the range from the first funcname line that matches *<funcname>*, up to the next funcname line. *:<funcname>* searches from the end of the previous *-L* range, if any, otherwise from the start of file. *^:<funcname>* searches from the start of file. The function names are determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in Section G.4.2, “gitattributes(5)”).

<revision range>

Limit the revisions to show. This can be either a single revision meaning show from the given revision and back, or it can be a range in the form "*<from>*..*<to>*" to show all revisions between *<from>* and back to *<to>*. Note, more advanced revision selection can be applied. For a more complete list of ways to spell object names, see Section G.4.14, “gitrevisions(7)”.

<path>…

Limit commits to the ones touching files in the given paths. Note, to avoid ambiguity with respect to revision names use "--" to separate the paths from any preceding options.

2. gitk-specific options

--argscmd=<command>

Command to be run each time gitk has to determine the revision range to show. The command is expected to print on its standard output a list of additional revisions to be shown, one per line. Use this instead of explicitly specifying a *<revision-range>* if the set of commits to show may vary between refreshes.

--select-commit=<ref>

Select the specified commit after loading the graph. Default behavior is equivalent to specifying *--select-commit=HEAD*.

Examples

gitk v2.6.12.. include/scsi drivers/scsi

Show the changes since version *v2.6.12* that changed any file in the include/scsi or drivers/scsi subdirectories

gitk --since="2 weeks ago" -- gitk

Show the changes during the last two weeks to the file *gitk*. The "--" is necessary to avoid confusion with the **branch** named *gitk*

gitk --max-count=100 --all -- Makefile

Show at most 100 changes made to the file *Makefile*. Instead of only looking for changes in the current branch look in all branches.

Files

User configuration and preferences are stored at:

- *$XDG_CONFIG_HOME/git/gitk* if it exists, otherwise
- *$HOME/.gitk* if it exists

If neither of the above exist then *$XDG_CONFIG_HOME/git/gitk* is created and used by default. If *$XDG_CONFIG_HOME* is not set it defaults to *$HOME/.config* in all cases.

History

Gitk was the first graphical repository browser. It's written in tcl/tk.

gitk is actually maintained as an independent project, but stable versions are distributed as part of the Git suite for the convenience of end users.

gitk-git/ comes from Paul Mackerras's gitk project:

`git://ozlabs.org/~paulus/gitk`

SEE ALSO

qgit(1)

A repository browser written in C++ using Qt.

tig(1)

A minimal repository browser and Git tool output highlighter written in C using Ncurses.

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.9. gitmailmap(5)

2024-07-02
Revision History
2024-07-02

NAME

gitmailmap - Map author/committer names and/or E-Mail addresses

SYNOPSIS

$GIT_WORK_TREE/.mailmap

DESCRIPTION

If the file *.mailmap* exists at the toplevel of the repository, or at the location pointed to by the *mailmap.file* or *mailmap.blob* configuration options (see Section G.3.29, “git-config(1)”), it is used to map author and committer names and email addresses to canonical real names and email addresses.

SYNTAX

The # character begins a comment to the end of line, blank lines are ignored.

In the simple form, each line in the file consists of the canonical real name of an author, whitespace, and an email address used in the commit (enclosed by < and >) to map to the name. For example:

```
Proper Name <commit@email.xx>
```

The more complex forms are:

```
<proper@email.xx> <commit@email.xx>
```

which allows mailmap to replace only the email part of a commit, and:

```
Proper Name <proper@email.xx> <commit@email.xx>
```

which allows mailmap to replace both the name and the email of a commit matching the specified commit email address, and:

```
Proper Name <proper@email.xx> Commit Name <commit@email.xx>
```

which allows mailmap to replace both the name and the email of a commit matching both the specified commit name and email address.

Both E-Mails and names are matched case-insensitively. For example this would also match the *Commit Name <commit@email.xx>* above:

```
Proper Name <proper@email.xx> CoMmIt NaMe <CoMmIt@EmAiL.xX>
```

NOTES

Git does not follow symbolic links when accessing a *.mailmap* file in the working tree. This keeps behavior consistent when the file is accessed from the index or a tree versus from the filesystem.

EXAMPLES

Your history contains commits by two authors, Jane and Joe, whose names appear in the repository under several forms:

```
Joe Developer <joe@example.com>
Joe R. Developer <joe@example.com>
Jane Doe <jane@example.com>
Jane Doe <jane@laptop.(none)>
Jane D. <jane@desktop.(none)>
```

Now suppose that Joe wants his middle name initial used, and Jane prefers her family name fully spelled out. A *.mailmap* file to correct the names would look like:

```
Joe R. Developer <joe@example.com>
Jane Doe <jane@example.com>
Jane Doe <jane@desktop.(none)>
```

Note that there's no need to map the name for *<jane@laptop.(none)>* to only correct the names. However, leaving the obviously broken *<jane@laptop.(none)>* and *<jane@desktop.(none)>* E-Mails as-is is usually not what you want. A *.mailmap* file which also corrects those is:

```
Joe R. Developer <joe@example.com>
Jane Doe <jane@example.com> <jane@laptop.(none)>
Jane Doe <jane@example.com> <jane@desktop.(none)>
```

Finally, let's say that Joe and Jane shared an E-Mail address, but not a name, e.g. by having these two commits in the history generated by a bug reporting system. I.e. names appearing in history as:

```
Joe <bugs@example.com>
Jane <bugs@example.com>
```

A full *.mailmap* file which also handles those cases (an addition of two lines to the above example) would be:

```
Joe R. Developer <joe@example.com>
Jane Doe <jane@example.com> <jane@laptop.(none)>
Jane Doe <jane@example.com> <jane@desktop.(none)>
Joe R. Developer <joe@example.com> Joe <bugs@example.com>
Jane Doe <jane@example.com> Jane <bugs@example.com>
```

SEE ALSO

Section G.3.16, “git-check-mailmap(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.10. gitmodules(5)

2024-07-02
Revision History
2024-07-02

NAME

gitmodules - Defining submodule properties

SYNOPSIS

$GIT_WORK_TREE/.gitmodules

DESCRIPTION

The *.gitmodules* file, located in the top-level directory of a Git working tree, is a text file with a syntax matching the requirements of Section G.3.29, “git-config(1)”.

The file contains one subsection per submodule, and the subsection value is the name of the submodule. The name is set to the path where the submodule has been added unless it was customized with the *--name* option of *git submodule add*. Each submodule section also contains the following required keys:

submodule.<name>.path

Defines the path, relative to the top-level directory of the Git working tree, where the submodule is expected to be checked out. The path name must not end with a /. All submodule paths must be unique within the *.gitmodules* file.

submodule.<name>.url

Defines a URL from which the submodule repository can be cloned. This may be either an absolute URL ready to be passed to Section G.3.24, “git-clone(1)” or (if it begins with ./ or ../) a location relative to the superproject's origin repository.

In addition, there are a number of optional keys:

submodule.<name>.update

Defines the default update procedure for the named submodule, i.e. how the submodule is updated by the *git submodule update* command in the superproject. This is only used by *git submodule init* to initialize the configuration variable of the same name. Allowed values here are *checkout*, *rebase*, *merge* or *none*, but not *!command* (for security reasons). See the description of the *update* command in Section G.3.141, “git-submodule(1)” for more details.

submodule.<name>.branch

A remote branch name for tracking updates in the upstream submodule. If the option is not specified, it defaults to the remote *HEAD*. A special value of . is used to indicate that the name of the branch in the submodule should be the same name as the current branch in the current repository. See the *--remote* documentation in Section G.3.141, “git-submodule(1)” for details.

submodule.<name>.fetchRecurseSubmodules

This option can be used to control recursive fetching of this submodule. If this option is also present in the submodule's entry in *.git/config* of the superproject, the setting there will override the one found in *.gitmodules*. Both settings can be overridden on the command line by using the *--[no-]recurse-submodules* option to *git fetch* and *git pull*.

submodule.<name>.ignore

Defines under what circumstances *git status* and the diff family show a submodule as modified. The following values are supported:

all

The submodule will never be considered modified (but will nonetheless show up in the output of status and commit when it has been staged).

dirty

All changes to the submodule's work tree will be ignored, only committed differences between the *HEAD* of the submodule and its recorded state in the superproject are taken into account.

untracked

Only untracked files in submodules will be ignored. Committed differences and modifications to tracked files will show up.

none

No modifications to submodules are ignored, all of committed differences, and modifications to tracked and untracked files are shown. This is the default option.

If this option is also present in the submodule's entry in *.git/config* of the superproject, the setting there will override the one found in *.gitmodules*.

Both settings can be overridden on the command line by using the *--ignore-submodules* option. The *git submodule* commands are not affected by this setting.

submodule.<name>.shallow

When set to true, a clone of this submodule will be performed as a shallow clone (with a history depth of 1) unless the user explicitly asks for a non-shallow clone.

NOTES

Git does not allow the *.gitmodules* file within a working tree to be a symbolic link, and will refuse to check out such a tree entry. This keeps behavior consistent when the file is accessed from the index or a tree versus from the filesystem, and helps Git reliably enforce security checks of the file contents.

EXAMPLES

Consider the following *.gitmodules* file:

```
[submodule "libfoo"]
        path = include/foo
        url = git://foo.com/git/lib.git

[submodule "libbar"]
        path = include/bar
        url = git://bar.com/git/lib.git
```

This defines two submodules, *libfoo* and *libbar*. These are expected to be checked out in the paths *include/foo* and *include/bar*, and for both submodules a URL is specified which can be used for cloning the submodules.

SEE ALSO

Section G.3.141, “git-submodule(1)”, Section G.4.15, “gitsubmodules(7)”, Section G.3.29, “git-config(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.11. gitnamespaces(7)

2024-07-02

Revision History

2024-07-02

NAME

gitnamespaces - Git namespaces

SYNOPSIS

GIT_NAMESPACE=<namespace> *git upload-pack*
GIT_NAMESPACE=<namespace> *git receive-pack*

DESCRIPTION

Git supports dividing the refs of a single repository into multiple namespaces, each of which has its own branches, tags, and HEAD. Git can expose each namespace as an independent repository to pull from and push to, while sharing the object store, and exposing all the refs to operations such as Section G.3.58, “git-gc(1)”.

Storing multiple repositories as namespaces of a single repository avoids storing duplicate copies of the same objects, such as when storing multiple branches of the same source. The alternates mechanism provides similar support for avoiding duplicates, but alternates do not prevent duplication between new objects added to the repositories without ongoing maintenance, while namespaces do.

To specify a namespace, set the *GIT_NAMESPACE* environment variable to the namespace. For each ref namespace, Git stores the corresponding refs in a directory under *refs/namespaces/*. For example, *GIT_NAMESPACE=foo* will store refs under *refs/namespaces/foo/*. You can also specify namespaces via the *--namespace* option to Section G.3.1, “git(1)”.

Note that namespaces which include a / will expand to a hierarchy of namespaces; for example, *GIT_NAMESPACE=foo/bar* will store refs under *refs/namespaces/foo/refs/namespaces/bar/*. This makes paths in *GIT_NAMESPACE* behave hierarchically, so that cloning with *GIT_NAMESPACE=foo/bar* produces the same result as cloning with *GIT_NAMESPACE=foo* and cloning from that repo with *GIT_NAMESPACE=bar*. It also avoids ambiguity with strange namespace paths such as *foo/refs/heads/*, which could otherwise generate directory/file conflicts within the *refs* directory.

Section G.3.151, “git-upload-pack(1)” and Section G.3.108, “git-receive-pack(1)” rewrite the names of refs as specified by *GIT_NAMESPACE*. git-upload-pack and git-receive-pack will ignore all references outside the specified namespace.

The smart HTTP server, Section G.3.65, “git-http-backend(1)”, will pass GIT_NAMESPACE through to the backend programs; see Section G.3.65, “git-http-backend(1)” for sample configuration to expose repository namespaces as repositories.

For a simple local test, you can use Section G.3.110, “git-remote-ext(1)”:

```
git clone ext::'git --namespace=foo %s /tmp/prefixed.git'
```

SECURITY

The fetch and push protocols are not designed to prevent one side from stealing data from the other repository that was not intended to be shared. If you have private data that you need to protect from a malicious peer, your best option is to store it in another repository. This applies to both clients and servers. In particular, namespaces on a server are not effective for read access control; you should only grant read access to a namespace to clients that you would trust with read access to the entire repository.

The known attack vectors are as follows:

1. The victim sends "have" lines advertising the IDs of objects it has that are not explicitly intended to be shared but can be used to optimize the transfer if the peer also has them. The attacker chooses an object ID X to steal and sends a ref to X, but isn't required to send the content of X because the victim already has it. Now the

victim believes that the attacker has X, and it sends the content of X back to the attacker later. (This attack is most straightforward for a client to perform on a server, by creating a ref to X in the namespace the client has access to and then fetching it. The most likely way for a server to perform it on a client is to "merge" X into a public branch and hope that the user does additional work on this branch and pushes it back to the server without noticing the merge.)

2. As in #1, the attacker chooses an object ID X to steal. The victim sends an object Y that the attacker already has, and the attacker falsely claims to have X and not Y, so the victim sends Y as a delta against X. The delta reveals regions of X that are similar to Y to the attacker.

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.12. gitremote-helpers(7)

2024-07-02

Revision History

2024-07-02

NAME

gitremote-helpers - Helper programs to interact with remote repositories

SYNOPSIS

git remote-<transport> <repository> [<URL>]

DESCRIPTION

Remote helper programs are normally not used directly by end users, but they are invoked by Git when it needs to interact with remote repositories Git does not support natively. A given helper will implement a subset of the capabilities documented here. When Git needs to interact with a repository using a remote helper, it spawns the helper as an independent process, sends commands to the helper's standard input, and expects results from the helper's standard output. Because a remote helper runs as an independent process from Git, there is no need to re-link Git to add a new helper, nor any need to link the helper with the implementation of Git.

Every helper must support the "capabilities" command, which Git uses to determine what other commands the helper will accept. Those other commands can be used to discover and update remote refs, transport objects between the object database and the remote repository, and update the local object store.

Git comes with a "curl" family of remote helpers, that handle various transport protocols, such as *git-remote-http*, *git-remote-https*, *git-remote-ftp* and *git-remote-ftps*. They implement the capabilities *fetch*, *option*, and *push*.

INVOCATION

Remote helper programs are invoked with one or (optionally) two arguments. The first argument specifies a remote repository as in Git; it is either the name of a configured remote or a URL. The second argument specifies a URL; it is usually of the form *<transport>://<address>*, but any arbitrary string is possible. The *GIT_DIR* environment variable is set up for the remote helper and can be used to determine where to store additional data or from which directory to invoke auxiliary Git commands.

When Git encounters a URL of the form *<transport>://<address>*, where *<transport>* is a protocol that it cannot handle natively, it automatically invokes *git remote-<transport>* with the full URL as the second argument. If such a URL is encountered directly on the command line, the first argument is the same as the second, and if it is encountered in a configured remote, the first argument is the name of that remote.

A URL of the form *<transport>::<address>* explicitly instructs Git to invoke *git remote-<transport>* with *<address>* as the second argument. If such a URL is encountered directly on the command line, the first argument is *<address>*, and if it is encountered in a configured remote, the first argument is the name of that remote.

Additionally, when a configured remote has *remote.<name>.vcs* set to *<transport>*, Git explicitly invokes *git remote-<transport>* with *<name>* as the first argument. If set, the second argument is *remote.<name>.url*; otherwise, the second argument is omitted.

INPUT FORMAT

Git sends the remote helper a list of commands on standard input, one per line. The first command is always the *capabilities* command, in response to which the remote helper must print a list of the capabilities it supports (see below) followed by a blank line. The response to the capabilities command determines what commands Git uses in the remainder of the command stream.

The command stream is terminated by a blank line. In some cases (indicated in the documentation of the relevant commands), this blank line is followed by a payload in some other protocol (e.g., the pack protocol), while in others it indicates the end of input.

1. Capabilities

Each remote helper is expected to support only a subset of commands. The operations a helper supports are declared to Git in the response to the *capabilities* command (see COMMANDS, below).

In the following, we list all defined capabilities and for each we list which commands a helper with that capability must provide.

1.1. Capabilities for Pushing

connect

Can attempt to connect to *git receive-pack* (for pushing), *git upload-pack*, etc for communication using git's native packfile protocol. This requires a bidirectional, full-duplex connection.

Supported commands: *connect*.

stateless-connect

Experimental; for internal use only. Can attempt to connect to a remote server for communication using git's wire-protocol version 2. See the documentation for the stateless-connect command for more information.

Supported commands: *stateless-connect*.

push

Can discover remote refs and push local commits and the history leading up to them to new or existing remote refs.

Supported commands: *list for-push*, *push*.

export

Can discover remote refs and push specified objects from a fast-import stream to remote refs.

Supported commands: *list for-push*, *export*.

If a helper advertises *connect*, Git will use it if possible and fall back to another capability if the helper requests so when connecting (see the *connect* command under COMMANDS). When choosing between *push* and *export*, Git prefers *push*. Other frontends may have some other order of preference.

no-private-update

When using the *refspec* capability, git normally updates the private ref on successful push. This update is disabled when the remote-helper declares the capability *no-private-update*.

1.2. Capabilities for Fetching

connect

Can try to connect to *git upload-pack* (for fetching), *git receive-pack*, etc for communication using the Git's native packfile protocol. This requires a bidirectional, full-duplex connection.

Supported commands: *connect*.

stateless-connect

Experimental; for internal use only. Can attempt to connect to a remote server for communication using git's wire-protocol version 2. See the documentation for the stateless-connect command for more information.

Supported commands: *stateless-connect*.

fetch

Can discover remote refs and transfer objects reachable from them to the local object store.

Supported commands: *list*, *fetch*.

import

Can discover remote refs and output objects reachable from them as a stream in fast-import format.

Supported commands: *list*, *import*.

check-connectivity

Can guarantee that when a clone is requested, the received pack is self contained and is connected.

get

Can use the *get* command to download a file from a given URI.

If a helper advertises *connect*, Git will use it if possible and fall back to another capability if the helper requests so when connecting (see the *connect* command under COMMANDS). When choosing between *fetch* and *import*, Git prefers *fetch*. Other frontends may have some other order of preference.

1.3. Miscellaneous capabilities

option

For specifying settings like *verbosity* (how much output to write to stderr) and *depth* (how much history is wanted in the case of a shallow clone) that affect how other commands are carried out.

refspec <refspec>

For remote helpers that implement *import* or *export*, this capability allows the refs to be constrained to a private namespace, instead of writing to refs/heads or refs/remotes directly. It is recommended that all importers providing the *import* capability use this. It's mandatory for *export*.

A helper advertising the capability *refspec refs/heads/*:refs/svn/origin/branches/** is saying that, when it is asked to *import refs/heads/topic*, the stream it outputs will update the *refs/svn/origin/branches/topic* ref.

This capability can be advertised multiple times. The first applicable refspec takes precedence. The left-hand of refspecs advertised with this capability must cover all refs reported by the list command. If no *refspec* capability is advertised, there is an implied *refspec *:**.

When writing remote-helpers for decentralized version control systems, it is advised to keep a local copy of the repository to interact with, and to let the private namespace refs point to this local repository, while the refs/remotes namespace is used to track the remote repository.

bidi-import

This modifies the *import* capability. The fast-import commands *cat-blob* and *ls* can be used by remote-helpers to retrieve information about blobs and trees that already exist in fast-import's memory. This requires a channel from fast-import to the remote-helper. If it is advertised in addition to "import", Git establishes a pipe from fast-import to the remote-helper's stdin. It follows that Git and fast-import are both connected to the remote-helper's stdin. Because Git can send multiple commands to the remote-helper it is required that helpers that use *bidi-import* buffer all *import* commands of a batch before sending data to fast-import. This is to prevent mixing commands and fast-import responses on the helper's stdin.

export-marks <file>

This modifies the *export* capability, instructing Git to dump the internal marks table to <file> when complete. For details, read up on *--export-marks=<file>* in Section G.3.46, “git-fast-export(1)”.

import-marks <file>

This modifies the *export* capability, instructing Git to load the marks specified in <file> before processing any input. For details, read up on *--import-marks=<file>* in Section G.3.46, “git-fast-export(1)”.

signed-tags

This modifies the *export* capability, instructing Git to pass *--signed-tags=verbatim* to Section G.3.46, “git-fast-export(1)”. In the absence of this capability, Git will use *--signed-tags=warn-strip*.

object-format

This indicates that the helper is able to interact with the remote side using an explicit hash algorithm extension.

COMMANDS

Commands are given by the caller on the helper's standard input, one per line.

capabilities

Lists the capabilities of the helper, one per line, ending with a blank line. Each capability may be preceded with *, which marks them mandatory for Git versions using the remote helper to understand. Any unknown mandatory capability is a fatal error.

Support for this command is mandatory.

list

Lists the refs, one per line, in the format "<value> <name> [<attr> …]". The value may be a hex sha1 hash, "@<dest>" for a symref, ":<keyword> <value>" for a key-value pair, or "?" to indicate that the helper could not get the value of the ref. A space-separated list of attributes follows the name; unrecognized attributes are ignored. The list ends with a blank line.

See REF LIST ATTRIBUTES for a list of currently defined attributes. See REF LIST KEYWORDS for a list of currently defined keywords.

Supported if the helper has the "fetch" or "import" capability.

list for-push

Similar to *list*, except that it is used if and only if the caller wants to the resulting ref list to prepare push commands. A helper supporting both push and fetch can use this to distinguish for which operation the output of *list* is going to be used, possibly reducing the amount of work that needs to be performed.

Supported if the helper has the "push" or "export" capability.

option <name> <value>

Sets the transport helper option <name> to <value>. Outputs a single line containing one of *ok* (option successfully set), *unsupported* (option not recognized) or *error* *<msg>* (option <name> is supported but <value> is not valid for it). Options should be set before other commands, and may influence the behavior of those commands.

See OPTIONS for a list of currently defined options.

Supported if the helper has the "option" capability.

fetch <sha1> <name>

Fetches the given object, writing the necessary objects to the database. Fetch commands are sent in a batch, one per line, terminated with a blank line. Outputs a single blank line when all fetch commands in the same batch are complete. Only objects which were reported in the output of *list* with a sha1 may be fetched this way.

Optionally may output a *lock* *<file>* line indicating the full path of a file under *$GIT_DIR/objects/pack* which is keeping a pack until refs can be suitably updated. The path must end with *.keep*. This is a mechanism to name a <pack,idx,keep> tuple by giving only the keep component. The kept pack will not be deleted by a concurrent repack, even though its objects may not be referenced until the fetch completes. The *.keep* file will be deleted at the conclusion of the fetch.

If option *check-connectivity* is requested, the helper must output *connectivity-ok* if the clone is self-contained and connected.

Supported if the helper has the "fetch" capability.

push +<src>:<dst>

Pushes the given local <src> commit or branch to the remote branch described by <dst>. A batch sequence of one or more *push* commands is terminated with a blank line (if there is only one reference to push, a single *push* command is followed by a blank line). For example, the following would be two batches of *push*, the first asking the remote-helper to push the local ref *master* to the remote ref *master* and the local *HEAD* to the remote *branch*, and the second asking to push ref *foo* to ref *bar* (forced update requested by the +).

```
push refs/heads/master:refs/heads/master
push HEAD:refs/heads/branch
\n
push +refs/heads/foo:refs/heads/bar
\n
```

Zero or more protocol options may be entered after the last *push* command, before the batch's terminating blank line.

When the push is complete, outputs one or more *ok* *<dst>* or *error* *<dst>* *<why>?* lines to indicate success or failure of each pushed ref. The status report output is terminated by a blank line. The option field <why> may be quoted in a C style string if it contains an LF.

Supported if the helper has the "push" capability.

import <name>

Produces a fast-import stream which imports the current value of the named ref. It may additionally import other refs as needed to construct the history efficiently. The script writes to a helper-specific private namespace. The value of the named ref should be written to a location in this namespace derived by applying the refspecs from the "refspec" capability to the name of the ref.

Especially useful for interoperability with a foreign versioning system.

Just like *push*, a batch sequence of one or more *import* is terminated with a blank line. For each batch of *import*, the remote helper should produce a fast-import stream terminated by a *done* command.

Note that if the *bidi-import* capability is used the complete batch sequence has to be buffered before starting to send data to fast-import to prevent mixing of commands and fast-import responses on the helper's stdin.

Supported if the helper has the "import" capability.

export

Instructs the remote helper that any subsequent input is part of a fast-import stream (generated by *git fast-export*) containing objects which should be pushed to the remote.

Especially useful for interoperability with a foreign versioning system.

The *export-marks* and *import-marks* capabilities, if specified, affect this command in so far as they are passed on to *git fast-export*, which then will load/store a table of marks for local objects. This can be used to implement for incremental operations.

Supported if the helper has the "export" capability.

connect <service>

Connects to given service. Standard input and standard output of helper are connected to specified service (git prefix is included in service name so e.g. fetching uses *git-upload-pack* as service) on remote side. Valid replies to this command are empty line (connection established), *fallback* (no smart transport support, fall back to dumb transports) and just exiting with error message printed (can't connect, don't bother trying to fall back). After line feed terminating the positive (empty) response, the output of service starts. After the connection ends, the remote helper exits.

Supported if the helper has the "connect" capability.

stateless-connect <service>

Experimental; for internal use only. Connects to the given remote service for communication using git's wire-protocol version 2. Valid replies to this command are empty line (connection established), *fallback* (no smart transport support, fall back to dumb transports) and just exiting with error message printed (can't connect, don't bother trying to fall back). After line feed terminating the positive (empty) response, the output of the service starts. Messages (both request and response) must consist of zero or more PKT-LINEs, terminating in a flush packet. Response messages will then have a response end packet after the flush packet to indicate the end of a response. The client must not expect the server to store any state in between request-response pairs. After the connection ends, the remote helper exits.

Supported if the helper has the "stateless-connect" capability.

get <uri> <path>

Downloads the file from the given *<uri>* to the given *<path>*. If *<path>.temp* exists, then Git assumes that the *.temp* file is a partial download from a previous attempt and will resume the download from that position.

If a fatal error occurs, the program writes the error message to stderr and exits. The caller should expect that a suitable error message has been printed if the child closes the connection without completing a valid response for the current command.

Additional commands may be supported, as may be determined from capabilities reported by the helper.

REF LIST ATTRIBUTES

The *list* command produces a list of refs in which each ref may be followed by a list of attributes. The following ref list attributes are defined.

unchanged

This ref is unchanged since the last import or fetch, although the helper cannot necessarily determine what value that produced.

REF LIST KEYWORDS

The *list* command may produce a list of key-value pairs. The following keys are defined.

object-format

The refs are using the given hash algorithm. This keyword is only used if the server and client both support the object-format extension.

OPTIONS

The following options are defined and (under suitable circumstances) set by Git if the remote helper has the *option* capability.

option verbosity <n>

Changes the verbosity of messages displayed by the helper. A value of 0 for <n> means that processes operate quietly, and the helper produces only error output. 1 is the default level of verbosity, and higher values of <n> correspond to the number of -v flags passed on the command line.

option progress {*true*|*false*}

Enables (or disables) progress messages displayed by the transport helper during a command.

option depth <depth>

Deepens the history of a shallow repository.

option deepen-since <timestamp>

Deepens the history of a shallow repository based on time.

option deepen-not <ref>

Deepens the history of a shallow repository excluding ref. Multiple options add up.

option deepen-relative {*true*|*false*}

Deepens the history of a shallow repository relative to current boundary. Only valid when used with "option depth".

option followtags {*true*|*false*}

If enabled the helper should automatically fetch annotated tag objects if the object the tag points at was transferred during the fetch command. If the tag is not fetched by the helper a second fetch command will usually be sent to ask for the tag specifically. Some helpers may be able to use this option to avoid a second network connection.

option dry-run {*true*|*false*}: If true, pretend the operation completed successfully, but don't actually change any repository data. For most helpers this only applies to the *push*, if supported.

option servpath <c-style-quoted-path>

Sets service path (--upload-pack, --receive-pack etc.) for next connect. Remote helper may support this option, but must not rely on this option being set before connect request occurs.

option check-connectivity {*true*|*false*}

Request the helper to check connectivity of a clone.

option force {*true*|*false*}

Request the helper to perform a force update. Defaults to *false*.

option cloning {*true*|*false*}

Notify the helper this is a clone request (i.e. the current repository is guaranteed empty).

option update-shallow {*true*|*false*}

Allow to extend .git/shallow if the new refs require it.

option pushcert {*true*|*false*}

GPG sign pushes.

option push-option <string>

Transmit <string> as a push option. As the push option must not contain LF or NUL characters, the string is not encoded.

option from-promisor {*true*|*false*}

Indicate that these objects are being fetched from a promisor.

option no-dependents {*true*|*false*}

Indicate that only the objects wanted need to be fetched, not their dependents.

option atomic {*true*|*false*}

When pushing, request the remote server to update refs in a single atomic transaction. If successful, all refs will be updated, or none will. If the remote side does not support this capability, the push will fail.

option object-format true

Indicate that the caller wants hash algorithm information to be passed back from the remote. This mode is used when fetching refs.

SEE ALSO

Section G.3.112, “git-remote(1)”

Section G.3.110, “git-remote-ext(1)”

Section G.3.111, “git-remote-fd(1)”

Section G.3.47, “git-fast-import(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.13. gitrepository-layout(5)

2024-07-02
Revision History
2024-07-02

NAME

gitrepository-layout - Git Repository Layout

SYNOPSIS

$GIT_DIR/*

DESCRIPTION

A Git repository comes in two different flavours:

- a *.git* directory at the root of the working tree;
- a *<project>.git* directory that is a *bare* repository (i.e. without its own working tree), that is typically used for exchanging histories with others by pushing into it and fetching from it.

Note: Also you can have a plain text file *.git* at the root of your working tree, containing *gitdir: <path>* to point at the real directory that has the repository. This mechanism is called a *gitfile* and is usually managed via the *git submodule* and *git worktree* commands. It is often used for a working tree of a submodule checkout, to allow you in the containing superproject to *git checkout* a branch that does not have the submodule. The *checkout* has to remove the entire submodule working tree, without losing the submodule repository.

These things may exist in a Git repository.

objects

Object store associated with this repository. Usually an object store is self sufficient (i.e. all the objects that are referred to by an object found in it are also found in it), but there are a few ways to violate it.

1. You could have an incomplete but locally usable repository by creating a shallow clone. See Section G.3.24, “git-clone(1)”.
2. You could be using the *objects/info/alternates* or *$GIT_ALTERNATE_OBJECT_DIRECTORIES* mechanisms to *borrow* objects from other object stores. A repository with this kind of incomplete object store is not suitable to be published for use with dumb transports but otherwise is OK as long as *objects/info/alternates* points at the object stores it borrows from.

   This directory is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/objects" will be used instead.

objects/[0-9a-f][0-9a-f]

A newly created object is stored in its own file. The objects are splayed over 256 subdirectories using the first two characters of the sha1 object name to keep the number of directory entries in *objects* itself to a manageable number. Objects found here are often called *unpacked* (or *loose*) objects.

objects/pack

Packs (files that store many objects in compressed form, along with index files to allow them to be randomly accessed) are found in this directory.

objects/info

Additional information about the object store is recorded in this directory.

objects/info/packs

This file is to help dumb transports discover what packs are available in this object store. Whenever a pack is added or removed, *git update-server-info* should be run to keep this file up to date if the repository is published for dumb transports. *git repack* does this by default.

objects/info/alternates

This file records paths to alternate object stores that this object store borrows objects from, one pathname per line. Note that not only native Git tools use it locally, but the HTTP fetcher also tries to use it remotely; this will usually work if you have relative paths (relative to the object database, not to the repository!) in your alternates file, but it will not work if you use absolute paths unless the absolute path in filesystem and web URL is the same. See also *objects/info/http-alternates*.

objects/info/http-alternates

This file records URLs to alternate object stores that this object store borrows objects from, to be used when the repository is fetched over HTTP.

refs

References are stored in subdirectories of this directory. The *git prune* command knows to preserve objects reachable from refs found in this directory and its subdirectories. This directory is ignored (except refs/bisect, refs/rewritten and refs/worktree) if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/refs" will be used instead.

refs/heads/*name*

records tip-of-the-tree commit objects of branch *name*

refs/tags/*name*

records any object name (not necessarily a commit object, or a tag object that points at a commit object).

refs/remotes/*name*

records tip-of-the-tree commit objects of branches copied from a remote repository.

refs/replace/<*obj-sha1*>

records the SHA-1 of the object that replaces <*obj-sha1*>. This is similar to info/grafts and is internally used and maintained by Section G.3.114, “git-replace(1)”. Such refs can be exchanged between repositories while grafts are not.

packed-refs

records the same information as refs/heads/, refs/tags/, and friends record in a more efficient way. See Section G.3.98, “git-pack-refs(1)”. This file is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/packed-refs" will be used instead.

HEAD

A symref (see glossary) to the *refs/heads/* namespace describing the currently active branch. It does not mean much if the repository is not associated with any working tree (i.e. a *bare* repository), but a valid Git repository **must** have the HEAD file; some porcelains may use it to guess the designated "default" branch of the repository (usually *master*). It is legal if the named branch *name* does not (yet) exist. In some legacy setups, it is a symbolic link instead of a symref that points at the current branch.

HEAD can also record a specific commit directly, instead of being a symref to point at the current branch. Such a state is often called *detached HEAD*. See Section G.3.19, “git-checkout(1)” for details.

config

Repository specific configuration file. This file is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/config" will be used instead.

config.worktree

Working directory specific configuration file for the main working directory in multiple working directory setup (see Section G.3.159, “git-worktree(1)”).

branches

A slightly deprecated way to store shorthands to be used to specify a URL to *git fetch*, *git pull* and *git push*. A file can be stored as *branches/*<*name*> and then *name* can be given to these commands in place of *repository* argument. See the REMOTES section in Section G.3.49, “git-fetch(1)” for details. This mechanism is legacy

and not likely to be found in modern repositories. This directory is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/branches" will be used instead.

hooks

Hooks are customization scripts used by various Git commands. A handful of sample hooks are installed when *git init* is run, but all of them are disabled by default. To enable, the *.sample* suffix has to be removed from the filename by renaming. Read Section G.4.7, “githooks(5)” for more details about each hook. This directory is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/hooks" will be used instead.

common

When multiple working trees are used, most of files in $GIT_DIR are per-worktree with a few known exceptions. All files under *common* however will be shared between all working trees.

index

The current index file for the repository. It is usually not found in a bare repository.

sharedindex.<SHA-1>

The shared index part, to be referenced by $GIT_DIR/index and other temporary index files. Only valid in split index mode.

info

Additional information about the repository is recorded in this directory. This directory is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/info" will be used instead.

info/refs

This file helps dumb transports discover what refs are available in this repository. If the repository is published for dumb transports, this file should be regenerated by *git update-server-info* every time a tag or branch is created or modified. This is normally done from the *hooks/update* hook, which is run by the *git-receive-pack* command when you *git push* into the repository.

info/grafts

This file records fake commit ancestry information, to pretend the set of parents a commit has is different from how the commit was actually created. One record per line describes a commit and its fake parents by listing their 40-byte hexadecimal object names separated by a space and terminated by a newline.

Note that the grafts mechanism is outdated and can lead to problems transferring objects between repositories; see Section G.3.114, “git-replace(1)” for a more flexible and robust system to do the same thing.

info/exclude

This file, by convention among Porcelains, stores the exclude pattern list. *.gitignore* is the per-directory ignore file. *git status*, *git add*, *git rm* and *git clean* look at it but the core Git commands do not look at it. See also: Section G.4.5, “gitignore(5)”.

info/attributes

Defines which attributes to assign to a path, similar to per-directory *.gitattributes* files. See also: Section G.4.2, “gitattributes(5)”.

info/sparse-checkout

This file stores sparse checkout patterns. See also: Section G.3.106, “git-read-tree(1)”.

remotes

Stores shorthands for URL and default refnames for use when interacting with remote repositories via *git fetch*, *git pull* and *git push* commands. See the REMOTES section in Section G.3.49, “git-fetch(1)” for details.

This mechanism is legacy and not likely to be found in modern repositories. This directory is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/remotes" will be used instead.

logs

Records of changes made to refs are stored in this directory. See Section G.3.148, “git-update-ref(1)” for more information. This directory is ignored (except logs/HEAD) if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/logs" will be used instead.

logs/refs/heads/*name*

Records all changes made to the branch tip named *name*.

logs/refs/tags/*name*

Records all changes made to the tag named *name*.

shallow

This is similar to *info/grafts* but is internally used and maintained by shallow clone mechanism. See *--depth* option to Section G.3.24, “git-clone(1)” and Section G.3.49, “git-fetch(1)”. This file is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/shallow" will be used instead.

commondir

If this file exists, $GIT_COMMON_DIR (see Section G.3.1, “git(1)”) will be set to the path specified in this file if it is not explicitly set. If the specified path is relative, it is relative to $GIT_DIR. The repository with commondir is incomplete without the repository pointed by "commondir".

modules

Contains the git-repositories of the submodules.

worktrees

Contains administrative data for linked working trees. Each subdirectory contains the working tree-related part of a linked working tree. This directory is ignored if $GIT_COMMON_DIR is set, in which case "$GIT_COMMON_DIR/worktrees" will be used instead.

worktrees/<id>/gitdir

A text file containing the absolute path back to the .git file that points to here. This is used to check if the linked repository has been manually removed and there is no need to keep this directory any more. The mtime of this file should be updated every time the linked repository is accessed.

worktrees/<id>/locked

If this file exists, the linked working tree may be on a portable device and not available. The presence of this file prevents *worktrees/<id>* from being pruned either automatically or manually by *git worktree prune*. The file may contain a string explaining why the repository is locked.

worktrees/<id>/config.worktree

Working directory specific configuration file.

Git Repository Format Versions

Every git repository is marked with a numeric version in the *core.repositoryformatversion* key of its *config* file. This version specifies the rules for operating on the on-disk repository data. An implementation of git which does not understand a particular version advertised by an on-disk repository MUST NOT operate on that repository; doing so risks not only producing wrong results, but actually losing data.

Because of this rule, version bumps should be kept to an absolute minimum. Instead, we generally prefer these strategies:

- bumping format version numbers of individual data files (e.g., index, packfiles, etc). This restricts the incompatibilities only to those files.
- introducing new data that gracefully degrades when used by older clients (e.g., pack bitmap files are ignored by older clients, which simply do not take advantage of the optimization they provide).

A whole-repository format version bump should only be part of a change that cannot be independently versioned. For instance, if one were to change the reachability rules for objects, or the rules for locking refs, that would require a bump of the repository format version.

Note that this applies only to accessing the repository's disk contents directly. An older client which understands only format *0* may still connect via *git://* to a repository using format *1*, as long as the server process understands format *1*.

The preferred strategy for rolling out a version bump (whether whole repository or for a single file) is to teach git to read the new format, and allow writing the new format with a config switch or command line option (for experimentation or for those who do not care about backwards compatibility with older gits). Then after a long period to allow the reading capability to become common, we may switch to writing the new format by default.

The currently defined format versions are:

1. Version *0*

This is the format defined by the initial version of git, including but not limited to the format of the repository directory, the repository configuration file, and the object and ref storage. Specifying the complete behavior of git is beyond the scope of this document.

2. Version *1*

This format is identical to version *0*, with the following exceptions:

1. When reading the *core.repositoryformatversion* variable, a git implementation which supports version 1 MUST also read any configuration keys found in the *extensions* section of the configuration file.
2. If a version-1 repository specifies any *extensions.** keys that the running git has not implemented, the operation MUST NOT proceed. Similarly, if the value of any known key is not understood by the implementation, the operation MUST NOT proceed.

Note that if no extensions are specified in the config file, then *core.repositoryformatversion* SHOULD be set to *0* (setting it to *1* provides no benefit, and makes the repository incompatible with older implementations of git).

This document will serve as the master list for extensions. Any implementation wishing to define a new extension should make a note of it here, in order to claim the name.

The defined extensions are:

2.1. *noop*

This extension does not change git's behavior at all. It is useful only for testing format-1 compatibility.

2.2. *preciousObjects*

When the config key *extensions.preciousObjects* is set to *true*, objects in the repository MUST NOT be deleted (e.g., by *git-prune* or *git repack -d*).

2.3. *partialClone*

When the config key *extensions.partialClone* is set, it indicates that the repo was created with a partial clone (or later performed a partial fetch) and that the remote may have omitted sending certain unwanted objects. Such a

remote is called a "promisor remote" and it promises that all such omitted objects can be fetched from it in the future.

The value of this key is the name of the promisor remote.

2.4. *worktreeConfig*

If set, by default "git config" reads from both "config" and "config.worktree" files from GIT_DIR in that order. In multiple working directory mode, "config" file is shared while "config.worktree" is per-working directory (i.e., it's in GIT_COMMON_DIR/worktrees/<id>/config.worktree)

2.5. *refStorage*

Specifies the file format for the ref database. The valid values are *files* (loose references with a packed-refs file) and *reftable* (see Documentation/technical/reftable.txt).

SEE ALSO

Section G.3.71, “git-init(1)”, Section G.3.24, “git-clone(1)”, Section G.3.49, “git-fetch(1)”, Section G.3.98, “git-pack-refs(1)”, Section G.3.58, “git-gc(1)”, Section G.3.19, “git-checkout(1)”, Section G.4.19, “gitglossary(7)”, *The Git User's Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html]

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.14. gitrevisions(7)

2024-07-02
Revision History
2024-07-02

NAME

gitrevisions - Specifying revisions and ranges for Git

SYNOPSIS

gitrevisions

DESCRIPTION

Many Git commands take revision parameters as arguments. Depending on the command, they denote a specific commit or, for commands which walk the revision graph (such as Section G.3.74, “git-log(1)”), all commits which are reachable from that commit. For commands that walk the revision graph one can also specify a range of revisions explicitly.

In addition, some Git commands (such as Section G.3.134, “git-show(1)” and Section G.3.103, “git-push(1)”) can also take revision parameters which denote other objects than commits, e.g. blobs ("files") or trees ("directories of files").

SPECIFYING REVISIONS

A revision parameter <*rev*> typically, but not necessarily, names a commit object. It uses what is called an *extended SHA-1* syntax. Here are various ways to spell object names. The ones listed near the end of this list name trees and blobs contained in a commit.

> **Note**
>
> This document shows the "raw" syntax as seen by git. The shell and other UIs might require additional quoting to protect special characters and to avoid word splitting.

<sha1>, e.g. *dae86e1950b1277e545cee180551750029cfe735*, *dae86e*

The full SHA-1 object name (40-byte hexadecimal string), or a leading substring that is unique within the repository. E.g. dae86e1950b1277e545cee180551750029cfe735 and dae86e both name the same commit object if there is no other object in your repository whose object name starts with dae86e.

<describeOutput>, e.g. *v1.7.4.2-679-g3bee7fb*

Output from *git describe*; i.e. a closest tag, optionally followed by a dash and a number of commits, followed by a dash, a *g*, and an abbreviated object name.

<refname>, e.g. *master*, *heads/master*, *refs/heads/master*

A symbolic ref name. E.g. *master* typically means the commit object referenced by *refs/heads/master*. If you happen to have both *heads/master* and *tags/master*, you can explicitly say *heads/master* to tell Git which one you mean. When ambiguous, a *<refname>* is disambiguated by taking the first match in the following rules:

1. If *$GIT_DIR/<refname>* exists, that is what you mean (this is usually useful only for *HEAD*, *FETCH_HEAD*, *ORIG_HEAD*, *MERGE_HEAD*, *REBASE_HEAD*, *REVERT_HEAD*, *CHERRY_PICK_HEAD*, *BISECT_HEAD* and *AUTO_MERGE*);

2. otherwise, *refs/<refname>* if it exists;

3. otherwise, *refs/tags/<refname>* if it exists;

4. otherwise, *refs/heads/<refname>* if it exists;

5. otherwise, *refs/remotes/<refname>* if it exists;

6. otherwise, *refs/remotes/<refname>/HEAD* if it exists.

HEAD

names the commit on which you based the changes in the working tree.

FETCH_HEAD

records the branch which you fetched from a remote repository with your last *git fetch* invocation.

ORIG_HEAD

is created by commands that move your *HEAD* in a drastic way (*git am*, *git merge*, *git rebase*, *git reset*), to record the position of the *HEAD* before their operation, so that you can easily change the tip of the branch back to the state before you ran them.

MERGE_HEAD

records the commit(s) which you are merging into your branch when you run *git merge*.

REBASE_HEAD

during a rebase, records the commit at which the operation is currently stopped, either because of conflicts or an *edit* command in an interactive rebase.

REVERT_HEAD

records the commit which you are reverting when you run *git revert*.

CHERRY_PICK_HEAD

records the commit which you are cherry-picking when you run *git cherry-pick*.

BISECT_HEAD

records the current commit to be tested when you run *git bisect --no-checkout*.

AUTO_MERGE

records a tree object corresponding to the state the *ort* merge strategy wrote to the working tree when a merge operation resulted in conflicts.

Note that any of the *refs/** cases above may come either from the *$GIT_DIR/refs* directory or from the *$GIT_DIR/packed-refs* file. While the ref name encoding is unspecified, UTF-8 is preferred as some output processing may assume ref names in UTF-8.

@

@ alone is a shortcut for *HEAD*.

[<refname>]@{<date>}, e.g. *master@{yesterday}*, *HEAD@{5 minutes ago}*

A ref followed by the suffix @ with a date specification enclosed in a brace pair (e.g. *{yesterday}*, *{1 month 2 weeks 3 days 1 hour 1 second ago}* or *{1979-02-26 18:30:00}*) specifies the value of the ref at a prior point in time. This suffix may only be used immediately following a ref name and the ref must have an existing log (*$GIT_DIR/logs/<ref>*). Note that this looks up the state of your **local** ref at a given time; e.g., what was in your local *master* branch last week. If you want to look at commits made during certain times, see *--since* and *--until*.

<refname>@{<n>}, e.g. *master@{1}*

A ref followed by the suffix @ with an ordinal specification enclosed in a brace pair (e.g. *{1}*, *{15}*) specifies the n-th prior value of that ref. For example *master@{1}* is the immediate prior value of *master* while *master@{5}* is the 5th prior value of *master*. This suffix may only be used immediately following a ref name and the ref must have an existing log (*$GIT_DIR/logs/<refname>*).

@{<n>}, e.g. *@{1}*

You can use the @ construct with an empty ref part to get at a reflog entry of the current branch. For example, if you are on branch *blabla* then *@{1}* means the same as *blabla@{1}*.

@{-<n>}, e.g. *@{-1}*

The construct *@{-<n>}* means the <n>th branch/commit checked out before the current one.

[<branchname>]@{upstream}, e.g. *master@{upstream}*, *@{u}*

A branch B may be set up to build on top of a branch X (configured with *branch.<name>.merge*) at a remote R (configured with the branch X taken from remote R, typically found at *refs/remotes/R/X*.

[<branchname>]@{push}, e.g. *master@{push}*, *@{push}*

The suffix *@{push}* reports the branch "where we would push to" if *git push* were run while *branchname* was checked out (or the current *HEAD* if no branchname is specified). Like for *@{upstream}*, we report the remote-tracking branch that corresponds to that branch at the remote.

Here's an example to make it more clear:

```
$ git config push.default current
$ git config remote.pushdefault myfork
$ git switch -c mybranch origin/master

$ git rev-parse --symbolic-full-name @{upstream}
refs/remotes/origin/master
```

```
$ git rev-parse --symbolic-full-name @{push}
refs/remotes/myfork/mybranch
```

Note in the example that we set up a triangular workflow, where we pull from one location and push to another. In a non-triangular workflow, *@{push}* is the same as *@{upstream}*, and there is no need for it.

This suffix is also accepted when spelled in uppercase, and means the same thing no matter the case.

<rev>^[<n>], e.g. *HEAD^, v1.5.1^0*

A suffix ^ to a revision parameter means the first parent of that commit object. ^*<n>* means the <n>th parent (i.e. *<rev>^* is equivalent to *<rev>^1*). As a special rule, *<rev>^0* means the commit itself and is used when *<rev>* is the object name of a tag object that refers to a commit object.

<rev>~[<n>], e.g. *HEAD~, master~3*

A suffix ~ to a revision parameter means the first parent of that commit object. A suffix ~*<n>* to a revision parameter means the commit object that is the <n>th generation ancestor of the named commit object, following only the first parents. I.e. *<rev>~3* is equivalent to *<rev>^^^* which is equivalent to *<rev>^1^1^1*. See below for an illustration of the usage of this form.

<rev>^{<type>}, e.g. *v0.99.8^{commit}*

A suffix ^ followed by an object type name enclosed in brace pair means dereference the object at *<rev>* recursively until an object of type *<type>* is found or the object cannot be dereferenced anymore (in which case, barf). For example, if *<rev>* is a commit-ish, *<rev>^{commit}* describes the corresponding commit object. Similarly, if *<rev>* is a tree-ish, *<rev>^{tree}* describes the corresponding tree object. *<rev>^0* is a short-hand for *<rev>^{commit}*.

<rev>^{object} can be used to make sure *<rev>* names an object that exists, without requiring *<rev>* to be a tag, and without dereferencing *<rev>*; because a tag is already an object, it does not have to be dereferenced even once to get to an object.

<rev>^{tag} can be used to ensure that *<rev>* identifies an existing tag object.

<rev>^{}, e.g. *v0.99.8^{}*

A suffix ^ followed by an empty brace pair means the object could be a tag, and dereference the tag recursively until a non-tag object is found.

<rev>^{/<text>}, e.g. *HEAD^{/fix nasty bug}*

A suffix ^ to a revision parameter, followed by a brace pair that contains a text led by a slash, is the same as the *:/fix nasty bug* syntax below except that it returns the youngest matching commit which is reachable from the *<rev>* before ^.

:/<text>, e.g. *:/fix nasty bug*

A colon, followed by a slash, followed by a text, names a commit whose commit message matches the specified regular expression. This name returns the youngest matching commit which is reachable from any ref, including HEAD. The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. *:/^foo*. The special sequence *:/!* is reserved for modifiers to what is matched. *:/!-foo* performs a negative match, while *:/!!foo* matches a literal *!* character, followed by *foo*. Any other sequence beginning with *:/!* is reserved for now. Depending on the given text, the shell's word splitting rules might require additional quoting.

<rev>:<path>, e.g. *HEAD:README, master:./README*

A suffix *:* followed by a path names the blob or tree at the given path in the tree-ish object named by the part before the colon. A path starting with *./* or *../* is relative to the current working directory. The given path will be converted to be relative to the working tree's root directory. This is most useful to address a blob or tree from a commit or tree that has the same tree structure as the working tree.

:[<n>:]<path>, e.g. *:0:README*, *:README*

A colon, optionally followed by a stage number (0 to 3) and a colon, followed by a path, names a blob object in the index at the given path. A missing stage number (and the colon that follows it) names a stage 0 entry. During a merge, stage 1 is the common ancestor, stage 2 is the target branch's version (typically the current branch), and stage 3 is the version from the branch which is being merged.

Here is an illustration, by Jon Loeliger. Both commit nodes B and C are parents of commit node A. Parent commits are ordered left-to-right.

```
G   H   I   J
 \ /     \ /
  D   E   F
   \  |  / \
    \ | /   |
     \|/    |
      B     C
       \   /
        \ /
         A

A =      = A^0
B = A^   = A^1     = A~1
C =      = A^2
D = A^^  = A^1^1   = A~2
E = B^2  = A^^2
F = B^3  = A^^3
G = A^^^ = A^1^1^1 = A~3
H = D^2  = B^^2    = A^^^2  = A~2^2
I = F^   = B^3^    = A^^3^
J = F^2  = B^3^2   = A^^3^2
```

SPECIFYING RANGES

History traversing commands such as *git log* operate on a set of commits, not just a single commit.

For these commands, specifying a single revision, using the notation described in the previous section, means the set of commits *reachable* from the given commit.

Specifying several revisions means the set of commits reachable from any of the given commits.

A commit's reachable set is the commit itself and the commits in its ancestry chain.

There are several notations to specify a set of connected commits (called a "revision range"), illustrated below.

1. Commit Exclusions

^<*rev*> (caret) Notation

To exclude commits reachable from a commit, a prefix ^notation is used. E.g. *^r1 r2* means commits reachable from *r2* but exclude the ones reachable from *r1* (i.e. *r1* and its ancestors).

2. Dotted Range Notations

The .. (two-dot) Range Notation

The *^r1 r2* set operation appears so often that there is a shorthand for it. When you have two commits *r1* and *r2* (named according to the syntax explained in SPECIFYING REVISIONS above), you can ask for commits that are reachable from r2 excluding those that are reachable from r1 by *^r1 r2* and it can be written as *r1..r2*.

The ... (three-dot) Symmetric Difference Notation

A similar notation *r1...r2* is called symmetric difference of *r1* and *r2* and is defined as *r1 r2 --not $(git merge-base --all r1 r2)*. It is the set of commits that are reachable from either one of *r1* (left side) or *r2* (right side) but not from both.

In these two shorthand notations, you can omit one end and let it default to HEAD. For example, *origin..* is a shorthand for *origin..HEAD* and asks "What did I do since I forked from the origin branch?" Similarly, *..origin* is a shorthand for *HEAD..origin* and asks "What did the origin do since I forked from them?" Note that .. would mean *HEAD..HEAD* which is an empty range that is both reachable and unreachable from HEAD.

Commands that are specifically designed to take two distinct ranges (e.g. "git range-diff R1 R2" to compare two ranges) do exist, but they are exceptions. Unless otherwise noted, all "git" commands that operate on a set of commits work on a single revision range. In other words, writing two "two-dot range notation" next to each other, e.g.

```
$ git log A..B C..D
```

does **not** specify two revision ranges for most commands. Instead it will name a single connected set of commits, i.e. those that are reachable from either B or D but are reachable from neither A or C. In a linear history like this:

```
---A---B---o---o---C---D
```

because A and B are reachable from C, the revision range specified by these two dotted ranges is a single commit D.

3. Other <rev>^ Parent Shorthand Notations

Three other shorthands exist, particularly useful for merge commits, for naming a set that is formed by a commit and its parent commits.

The *r1^@* notation means all parents of *r1*.

The *r1^!* notation includes commit *r1* but excludes all of its parents. By itself, this notation denotes the single commit *r1*.

The *<rev>^-[<n>]* notation includes *<rev>* but excludes the <n>th parent (i.e. a shorthand for *<rev>^<n>..<rev>*), with *<n>* = 1 if not given. This is typically useful for merge commits where you can just pass *<commit>^-* to get all the commits in the branch that was merged in merge commit *<commit>* (including *<commit>* itself).

While *<rev>^<n>* was about specifying a single commit parent, these three notations also consider its parents. For example you can say *HEAD^2^@*, however you cannot say *HEAD^@^2*.

Revision Range Summary

<rev>

Include commits that are reachable from <rev> (i.e. <rev> and its ancestors).

^*<rev>*

Exclude commits that are reachable from <rev> (i.e. <rev> and its ancestors).

<rev1>..<rev2>

Include commits that are reachable from <rev2> but exclude those that are reachable from <rev1>. When either <rev1> or <rev2> is omitted, it defaults to *HEAD*.

<rev1>...<rev2>

Include commits that are reachable from either <rev1> or <rev2> but exclude those that are reachable from both. When either <rev1> or <rev2> is omitted, it defaults to *HEAD*.

<rev>^@, e.g. *HEAD^@*

A suffix ^ followed by an at sign is the same as listing all parents of *<rev>* (meaning, include anything reachable from its parents, but not the commit itself).

<rev>^!, e.g. *HEAD^!*

A suffix ^ followed by an exclamation mark is the same as giving commit *<rev>* and all its parents prefixed with ^ to exclude them (and their ancestors).

<rev>^-<n>, e.g. *HEAD^-*, *HEAD^-2*

Equivalent to *<rev>^<n>..<rev>*, with *<n>* = 1 if not given.

Here are a handful of examples using the Loeliger illustration above, with each step in the notation's expansion and selection carefully spelt out:

```
Args   Expanded arguments    Selected commits
D                            G H D
D F                          G H I J D F
^G D                         H D
^D B                         E I J F B
^D B C                       E I J F B C
C                            I J F C
B..C   = ^B C                C
B...C  = B ^F C              G H D E B C
B^-    = B^..B
       = ^B^1 B              E I J F B
C^@    = C^1
       = F                   I J F
B^@    = B^1 B^2 B^3
       = D E F               D G H E F I J
C^!    = C ^C^@
       = C ^C^1
       = C ^F                C
B^!    = B ^B^@
       = B ^B^1 ^B^2 ^B^3
       = B ^D ^E ^F          B
F^! D  = F ^I ^J D           G H D F
```

SEE ALSO

Section G.3.121, “git-rev-parse(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.15. gitsubmodules(7)

2024-07-02

Revision History

2024-07-02

NAME

gitsubmodules - Mounting one repository inside another

SYNOPSIS

`.gitmodules, $GIT_DIR/config`

`git submodule`

```
git <command> --recurse-submodules
```

DESCRIPTION

A submodule is a repository embedded inside another repository. The submodule has its own history; the repository it is embedded in is called a superproject.

On the filesystem, a submodule usually (but not always - see FORMS below) consists of (i) a Git directory located under the *$GIT_DIR/modules/* directory of its superproject, (ii) a working directory inside the superproject's working directory, and a *.git* file at the root of the submodule's working directory pointing to (i).

Assuming the submodule has a Git directory at *$GIT_DIR/modules/foo/* and a working directory at *path/to/bar/*, the superproject tracks the submodule via a *gitlink* entry in the tree at *path/to/bar* and an entry in its *.gitmodules* file (see Section G.4.10, “gitmodules(5)”) of the form *submodule.foo.path = path/to/bar*.

The *gitlink* entry contains the object name of the commit that the superproject expects the submodule's working directory to be at.

The section *submodule.foo.** in the *.gitmodules* file gives additional hints to Git's porcelain layer. For example, the *submodule.foo.url* setting specifies where to obtain the submodule.

Submodules can be used for at least two different use cases:

1. Using another project while maintaining independent history. Submodules allow you to contain the working tree of another project within your own working tree while keeping the history of both projects separate. Also, since submodules are fixed to an arbitrary version, the other project can be independently developed without affecting the superproject, allowing the superproject project to fix itself to new versions only when desired.

2. Splitting a (logically single) project into multiple repositories and tying them back together. This can be used to overcome current limitations of Git's implementation to have finer grained access:

   - Size of the Git repository: In its current form Git scales up poorly for large repositories containing content that is not compressed by delta computation between trees. For example, you can use submodules to hold large binary assets and these repositories can be shallowly cloned such that you do not have a large history locally.

   - Transfer size: In its current form Git requires the whole working tree present. It does not allow partial trees to be transferred in fetch or clone. If the project you work on consists of multiple repositories tied together as submodules in a superproject, you can avoid fetching the working trees of the repositories you are not interested in.

   - Access control: By restricting user access to submodules, this can be used to implement read/write policies for different users.

The configuration of submodules

Submodule operations can be configured using the following mechanisms (from highest to lowest precedence):

- The command line for those commands that support taking submodules as part of their pathspecs. Most commands have a boolean flag *--recurse-submodules* which specifies whether to recurse into submodules. Examples are *grep* and *checkout*. Some commands take enums, such as *fetch* and *push*, where you can specify how submodules are affected.

- The configuration inside the submodule. This includes *$GIT_DIR/config* in the submodule, but also settings in the tree such as a *.gitattributes* or *.gitignore* files that specify behavior of commands inside the submodule.

  For example an effect from the submodule's *.gitignore* file would be observed when you run *git status --ignore-submodules=none* in the superproject. This collects information from the submodule's working directory by running *status* in the submodule while paying attention to the *.gitignore* file of the submodule.

  The submodule's *$GIT_DIR/config* file would come into play when running *git push --recurse-submodules=check* in the superproject, as this would check if the submodule has any changes not published to any remote. The remotes are configured in the submodule as usual in the *$GIT_DIR/config* file.

- The configuration file *$GIT_DIR/config* in the superproject. Git only recurses into active submodules (see "ACTIVE SUBMODULES" section below).

  If the submodule is not yet initialized, then the configuration inside the submodule does not exist yet, so where to obtain the submodule from is configured here for example.

- The *.gitmodules* file inside the superproject. A project usually uses this file to suggest defaults for the upstream collection of repositories for the mapping that is required between a submodule's name and its path.

  This file mainly serves as the mapping between the name and path of submodules in the superproject, such that the submodule's Git directory can be located.

  If the submodule has never been initialized, this is the only place where submodule configuration is found. It serves as the last fallback to specify where to obtain the submodule from.

FORMS

Submodules can take the following forms:

- The basic form described in DESCRIPTION with a Git directory, a working directory, a *gitlink*, and a *.gitmodules* entry.

- "Old-form" submodule: A working directory with an embedded *.git* directory, and the tracking *gitlink* and *.gitmodules* entry in the superproject. This is typically found in repositories generated using older versions of Git.

  It is possible to construct these old form repositories manually.

  When deinitialized or deleted (see below), the submodule's Git directory is automatically moved to *$GIT_DIR/modules/<name>/* of the superproject.

- Deinitialized submodule: A *gitlink*, and a *.gitmodules* entry, but no submodule working directory. The submodule's Git directory may be there as after deinitializing the Git directory is kept around. The directory which is supposed to be the working directory is empty instead.

  A submodule can be deinitialized by running *git submodule deinit*. Besides emptying the working directory, this command only modifies the superproject's *$GIT_DIR/config* file, so the superproject's history is not affected. This can be undone using *git submodule init*.

- Deleted submodule: A submodule can be deleted by running *git rm <submodule-path> && git commit*. This can be undone using *git revert*.

  The deletion removes the superproject's tracking data, which are both the *gitlink* entry and the section in the *.gitmodules* file. The submodule's working directory is removed from the file system, but the Git directory is kept around as it to make it possible to checkout past commits without requiring fetching from another repository.

  To completely remove a submodule, manually delete *$GIT_DIR/modules/<name>/*.

ACTIVE SUBMODULES

A submodule is considered active,

1. if *submodule.<name>.active* is set to *true*

   or

2. if the submodule's path matches the pathspec in *submodule.active*

   or

3. if *submodule.<name>.url* is set.

and these are evaluated in this order.

For example:

```
[submodule "foo"]
  active = false
  url = https://example.org/foo
[submodule "bar"]
  active = true
  url = https://example.org/bar
[submodule "baz"]
  url = https://example.org/baz
```

In the above config only the submodules *bar* and *baz* are active, *bar* due to (1) and *baz* due to (3). *foo* is inactive because (1) takes precedence over (3)

Note that (3) is a historical artefact and will be ignored if the (1) and (2) specify that the submodule is not active. In other words, if we have a *submodule.<name>.active* set to *false* or if the submodule's path is excluded in the pathspec in *submodule.active*, the url doesn't matter whether it is present or not. This is illustrated in the example that follows.

```
[submodule "foo"]
  active = true
  url = https://example.org/foo
[submodule "bar"]
  url = https://example.org/bar
[submodule "baz"]
  url = https://example.org/baz
[submodule "bob"]
  ignore = true
[submodule]
  active = b*
  active = :(exclude) baz
```

In here all submodules except *baz* (foo, bar, bob) are active. *foo* due to its own active flag and all the others due to the submodule active pathspec, which specifies that any submodule starting with *b* except *baz* are also active, regardless of the presence of the .url field.

Workflow for a third party library

```
# Add a submodule
git submodule add <URL> <path>

# Occasionally update the submodule to a new version:
git -C <path> checkout <new-version>
git add <path>
git commit -m "update submodule to new version"

# See the list of submodules in a superproject
git submodule status

# See FORMS on removing submodules
```

Workflow for an artificially split repo

```
# Enable recursion for relevant commands, such that
# regular commands recurse into submodules by default
git config --global submodule.recurse true

# Unlike most other commands below, clone still needs
# its own recurse flag:
git clone --recurse <URL> <directory>
```

```
cd <directory>

# Get to know the code:
git grep foo
git ls-files --recurse-submodules
```

> **Note**
>
> *git ls-files* also requires its own *--recurse-submodules* flag.

```
# Get new code
git fetch
git pull --rebase

# Change worktree
git checkout
git reset
```

Implementation details

When cloning or pulling a repository containing submodules the submodules will not be checked out by default; you can instruct *clone* to recurse into submodules. The *init* and *update* subcommands of *git submodule* will maintain submodules checked out and at an appropriate revision in your working tree. Alternatively you can set *submodule.recurse* to have *checkout* recurse into submodules (note that *submodule.recurse* also affects other Git commands, see Section G.3.29, “git-config(1)” for a complete list).

SEE ALSO

Section G.3.141, “git-submodule(1)”, Section G.4.10, “gitmodules(5)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.16. gitweb(1)

2024-07-02
Revision History
2024-07-02

NAME

gitweb - Git web interface (web frontend to Git repositories)

SYNOPSIS

To get started with gitweb, run Section G.3.72, “git-instaweb(1)” from a Git repository. This will configure and start your web server, and run a web browser pointing to gitweb.

DESCRIPTION

Gitweb provides a web interface to Git repositories. Its features include:

- Viewing multiple Git repositories with common root.
- Browsing every revision of the repository.
- Viewing the contents of files in the repository at any revision.
- Viewing the revision log of branches, history of files and directories, seeing what was changed, when, and by whom.

- Viewing the blame/annotation details of any file (if enabled).
- Generating RSS and Atom feeds of commits, for any branch. The feeds are auto-discoverable in modern web browsers.
- Viewing everything that was changed in a revision, and stepping through revisions one at a time, viewing the history of the repository.
- Finding commits whose commit messages match a given search term.

See https://repo.or.cz/w/git.git/tree/HEAD:/gitweb/ for gitweb source code, browsed using gitweb itself.

CONFIGURATION

Various aspects of gitweb's behavior can be controlled through the configuration file *gitweb_config.perl* or */etc/gitweb.conf*. See the Section G.4.17, “gitweb.conf(5)” for details.

1. Repositories

Gitweb can show information from one or more Git repositories. These repositories have to be all on local filesystem, and have to share a common repository root, i.e. be all under a single parent repository (but see also the "Advanced web server setup" section, "Webserver configuration with multiple projects' root" subsection).

```
our $projectroot = '/path/to/parent/directory';
```

The default value for *$projectroot* is */pub/git*. You can change it during building gitweb via the *GITWEB_PROJECTROOT* build configuration variable.

By default all Git repositories under *$projectroot* are visible and available to gitweb. The list of projects is generated by default by scanning the *$projectroot* directory for Git repositories (for object databases to be more exact; gitweb is not interested in a working area, and is best suited to showing "bare" repositories).

The name of the repository in gitweb is the path to its *$GIT_DIR* (its object database) relative to *$projectroot*. Therefore the repository $repo can be found at "$projectroot/$repo".

2. Projects list file format

Instead of having gitweb find repositories by scanning the filesystem starting from $projectroot, you can provide a pre-generated list of visible projects by setting *$projects_list* to point to a plain text file with a list of projects (with some additional info).

This file uses the following format:

- One record (for project / repository) per line; does not support line continuation (newline escaping).
- Leading and trailing whitespace are ignored.
- Whitespace separated fields; any run of whitespace can be used as field separator (rules for Perl's "*split(" ", $line)*").
- Fields use modified URI encoding, defined in RFC 3986, section 2.1 (Percent-Encoding), or rather "Query string encoding" (see https://en.wikipedia.org/wiki/Query_string#URL_encoding), the difference being that SP (" ") can be encoded as "+" (and therefore "+" has to be also percent-encoded).

  Reserved characters are: "%" (used for encoding), "+" (can be used to encode SPACE), all whitespace characters as defined in Perl, including SP, TAB and LF, (used to separate fields in a record).

- Currently recognized fields are:

  <repository path>

  path to repository GIT_DIR, relative to *$projectroot*

<repository owner>

displayed as repository owner, preferably full name, or email, or both

You can generate the projects list index file using the project_index action (the *TXT* link on projects list page) directly from gitweb; see also "Generating projects list using gitweb" section below.

Example contents:

```
foo.git        Joe+R+Hacker+<joe@example.com>
foo/bar.git    O+W+Ner+<owner@example.org>
```

By default this file controls only which projects are **visible** on projects list page (note that entries that do not point to correctly recognized Git repositories won't be displayed by gitweb). Even if a project is not visible on projects list page, you can view it nevertheless by hand-crafting a gitweb URL. By setting *$strict_export* configuration variable (see Section G.4.17, "gitweb.conf(5)") to true value you can allow viewing only of repositories also shown on the overview page (i.e. only projects explicitly listed in projects list file will be accessible).

3. Generating projects list using gitweb

We assume that GITWEB_CONFIG has its default Makefile value, namely *gitweb_config.perl*. Put the following in *gitweb_make_index.perl* file:

```
read_config_file("gitweb_config.perl");
$projects_list = $projectroot;
```

Then create the following script to get list of project in the format suitable for GITWEB_LIST build configuration variable (or *$projects_list* variable in gitweb config):

```
#!/bin/sh

export GITWEB_CONFIG="gitweb_make_index.perl"
export GATEWAY_INTERFACE="CGI/1.1"
export HTTP_ACCEPT="*/*"
export REQUEST_METHOD="GET"
export QUERY_STRING="a=project_index"

perl -- /var/www/cgi-bin/gitweb.cgi
```

Run this script and save its output to a file. This file could then be used as projects list file, which means that you can set *$projects_list* to its filename.

4. Controlling access to Git repositories

By default all Git repositories under *$projectroot* are visible and available to gitweb. You can however configure how gitweb controls access to repositories.

- As described in "Projects list file format" section, you can control which projects are **visible** by selectively including repositories in projects list file, and setting *$projects_list* gitweb configuration variable to point to it. With *$strict_export* set, projects list file can be used to control which repositories are **available** as well.

- You can configure gitweb to only list and allow viewing of the explicitly exported repositories, via *$export_ok* variable in gitweb config file; see Section G.4.17, "gitweb.conf(5)" manpage. If it evaluates to true, gitweb shows repositories only if this file named by *$export_ok* exists in its object database (if directory has the magic file named *$export_ok*).

  For example Section G.3.38, "git-daemon(1)" by default (unless *--export-all* option is used) allows pulling only for those repositories that have *git-daemon-export-ok* file. Adding

```
our $export_ok = "git-daemon-export-ok";
```

makes gitweb show and allow access only to those repositories that can be fetched from via *git://* protocol.

- Finally, it is possible to specify an arbitrary perl subroutine that will be called for each repository to determine if it can be exported. The subroutine receives an absolute path to the project (repository) as its only parameter (i.e. "$projectroot/$project").

  For example, if you use mod_perl to run the script, and have dumb HTTP protocol authentication configured for your repositories, you can use the following hook to allow access only if the user is authorized to read the files:

```
$export_auth_hook = sub {
        use Apache2::SubRequest ();
        use Apache2::Const -compile => qw(HTTP_OK);
        my $path = "$_[0]/HEAD";
        my $r    = Apache2::RequestUtil->request;
        my $sub  = $r->lookup_file($path);
        return $sub->filename eq $path
            && $sub->status == Apache2::Const::HTTP_OK;
};
```

5. Per-repository gitweb configuration

You can configure individual repositories shown in gitweb by creating file in the *GIT_DIR* of Git repository, or by setting some repo configuration variable (in *GIT_DIR/config*, see Section G.3.29, “git-config(1)”).

You can use the following files in repository:

README.html

A html file (HTML fragment) which is included on the gitweb project "summary" page inside <*div*> block element. You can use it for longer description of a project, to provide links (for example to project's homepage), etc. This is recognized only if XSS prevention is off (*$prevent_xss* is false, see Section G.4.17, “gitweb.conf(5)”); a way to include a README safely when XSS prevention is on may be worked out in the future.

description (or *gitweb.description*)

Short (shortened to *$projects_list_description_width* in the projects list page, which is 25 characters by default; see Section G.4.17, “gitweb.conf(5)”) single line description of a project (of a repository). Plain text file; HTML will be escaped. By default set to

```
Unnamed repository; edit this file to name it for gitweb.
```

from the template during repository creation, usually installed in */usr/share/git-core/templates/*. You can use the *gitweb.description* repo configuration variable, but the file takes precedence.

category (or *gitweb.category*)

Singe line category of a project, used to group projects if *$projects_list_group_categories* is enabled. By default (file and configuration variable absent), uncategorized projects are put in the *$project_list_default_category* category. You can use the *gitweb.category* repo configuration variable, but the file takes precedence.

The configuration variables *$projects_list_group_categories* and *$project_list_default_category* are described in Section G.4.17, “gitweb.conf(5)”

cloneurl (or multiple-valued *gitweb.url*)

File with repository URL (used for clone and fetch), one per line. Displayed in the project summary page. You can use multiple-valued *gitweb.url* repository configuration variable for that, but the file takes precedence.

This is per-repository enhancement / version of global prefix-based *@git_base_url_list* gitweb configuration variable (see Section G.4.17, “gitweb.conf(5)”).

gitweb.owner

You can use the *gitweb.owner* repository configuration variable to set repository's owner. It is displayed in the project list and summary page.

If it's not set, filesystem directory's owner is used (via GECOS field, i.e. real name field from **getpwuid**(3)) if *$projects_list* is unset (gitweb scans *$projectroot* for repositories); if *$projects_list* points to file with list of repositories, then project owner defaults to value from this file for given repository.

various *gitweb.** config variables (in config)

Read description of *%feature* hash for detailed list, and descriptions. See also "Configuring gitweb features" section in Section G.4.17, “gitweb.conf(5)”

ACTIONS, AND URLS

Gitweb can use path_info (component) based URLs, or it can pass all necessary information via query parameters. The typical gitweb URLs are broken down in to five components:

```
.../gitweb.cgi/<repo>/<action>/<revision>:/<path>?<arguments>
```

repo

The repository the action will be performed on.

All actions except for those that list all available projects, in whatever form, require this parameter.

action

The action that will be run. Defaults to *projects_list* if repo is not set, and to *summary* otherwise.

revision

Revision shown. Defaults to HEAD.

path

The path within the <repository> that the action is performed on, for those actions that require it.

arguments

Any arguments that control the behaviour of the action.

Some actions require or allow to specify two revisions, and sometimes even two pathnames. In most general form such path_info (component) based gitweb URL looks like this:

```
.../gitweb.cgi/<repo>/<action>/<revision-from>:/<path-from>..<revision-
to>:/<path-to>?<arguments>
```

Each action is implemented as a subroutine, and must be present in %actions hash. Some actions are disabled by default, and must be turned on via feature mechanism. For example to enable *blame* view add the following to gitweb configuration file:

```
$feature{'blame'}{'default'} = [1];
```

1. Actions:

The standard actions are:

project_list

Lists the available Git repositories. This is the default command if no repository is specified in the URL.

summary

Displays summary about given repository. This is the default command if no action is specified in URL, and only repository is specified.

heads , remotes

Lists all local or all remote-tracking branches in given repository.

The latter is not available by default, unless configured.

tags

List all tags (lightweight and annotated) in given repository.

blob , tree

Shows the files and directories in a given repository path, at given revision. This is default command if no action is specified in the URL, and path is given.

blob_plain

Returns the raw data for the file in given repository, at given path and revision. Links to this action are marked *raw*.

blobdiff

Shows the difference between two revisions of the same file.

blame , blame_incremental

Shows the blame (also called annotation) information for a file. On a per line basis it shows the revision in which that line was last changed and the user that committed the change. The incremental version (which if configured is used automatically when JavaScript is enabled) uses Ajax to incrementally add blame info to the contents of given file.

This action is disabled by default for performance reasons.

commit , commitdiff

Shows information about a specific commit in a repository. The *commit* view shows information about commit in more detail, the *commitdiff* action shows changeset for given commit.

patch

Returns the commit in plain text mail format, suitable for applying with Section G.3.3, “git-am(1)”.

tag

Display specific annotated tag (tag object).

log , shortlog

Shows log information (commit message or just commit subject) for a given branch (starting from given revision).

The *shortlog* view is more compact; it shows one commit per line.

history

Shows history of the file or directory in a given repository path, starting from given revision (defaults to HEAD, i.e. default branch).

This view is similar to *shortlog* view.

rss , atom

Generates an RSS (or Atom) feed of changes to repository.

WEBSERVER CONFIGURATION

This section explains how to configure some common webservers to run gitweb. In all cases, */path/to/gitweb* in the examples is the directory you ran installed gitweb in, and contains *gitweb_config.perl*.

If you've configured a web server that isn't listed here for gitweb, please send in the instructions so they can be included in a future release.

1. Apache as CGI

Apache must be configured to support CGI scripts in the directory in which gitweb is installed. Let's assume that it is */var/www/cgi-bin* directory.

```
ScriptAlias /cgi-bin/ "/var/www/cgi-bin/"

<Directory "/var/www/cgi-bin">
    Options Indexes FollowSymlinks ExecCGI
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

With that configuration the full path to browse repositories would be:

```
http://server/cgi-bin/gitweb.cgi
```

2. Apache with mod_perl, via ModPerl::Registry

You can use mod_perl with gitweb. You must install Apache::Registry (for mod_perl 1.x) or ModPerl::Registry (for mod_perl 2.x) to enable this support.

Assuming that gitweb is installed to */var/www/perl*, the following Apache configuration (for mod_perl 2.x) is suitable.

```
Alias /perl "/var/www/perl"

<Directory "/var/www/perl">
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlOptions +ParseHeaders
    Options Indexes FollowSymlinks +ExecCGI
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

With that configuration the full path to browse repositories would be:

```
http://server/perl/gitweb.cgi
```

3. Apache with FastCGI

Gitweb works with Apache and FastCGI. First you need to rename, copy or symlink gitweb.cgi to gitweb.fcgi. Let's assume that gitweb is installed in */usr/share/gitweb* directory. The following Apache configuration is suitable (UNTESTED!)

```
FastCgiServer /usr/share/gitweb/gitweb.cgi
ScriptAlias /gitweb /usr/share/gitweb/gitweb.cgi

Alias /gitweb/static /usr/share/gitweb/static
<Directory /usr/share/gitweb/static>
    SetHandler default-handler
</Directory>
```

With that configuration the full path to browse repositories would be:

```
http://server/gitweb
```

ADVANCED WEB SERVER SETUP

All of those examples use request rewriting, and need *mod_rewrite* (or equivalent; examples below are written for Apache).

1. Single URL for gitweb and for fetching

If you want to have one URL for both gitweb and your *http://* repositories, you can configure Apache like this:

```
<VirtualHost *:80>
    ServerName    git.example.org
    DocumentRoot  /pub/git
    SetEnv        GITWEB_CONFIG   /etc/gitweb.conf

    # turning on mod rewrite
    RewriteEngine on

    # make the front page an internal rewrite to the gitweb script
    RewriteRule ^/$  /cgi-bin/gitweb.cgi

    # make access for "dumb clients" work
    RewriteRule ^/(.*\.git/(?!/?(HEAD|info|objects|refs)).*)?$ \
                /cgi-bin/gitweb.cgi%{REQUEST_URI}  [L,PT]
</VirtualHost>
```

The above configuration expects your public repositories to live under */pub/git* and will serve them as *http://git.domain.org/dir-under-pub-git*, both as clonable Git URL and as browsable gitweb interface. If you then start your Section G.3.38, “git-daemon(1)” with *--base-path=/pub/git --export-all* then you can even use the *git://* URL with exactly the same path.

Setting the environment variable *GITWEB_CONFIG* will tell gitweb to use the named file (i.e. in this example */etc/gitweb.conf*) as a configuration for gitweb. You don't really need it in above example; it is required only if your configuration file is in different place than built-in (during compiling gitweb) *gitweb_config.perl* or */etc/gitweb.conf*. See Section G.4.17, “gitweb.conf(5)” for details, especially information about precedence rules.

If you use the rewrite rules from the example you **might** also need something like the following in your gitweb configuration file (*/etc/gitweb.conf* following example):

```
@stylesheets = ("/some/absolute/path/gitweb.css");
$my_uri    = "/";
$home_link = "/";
$per_request_config = 1;
```

Nowadays though gitweb should create HTML base tag when needed (to set base URI for relative links), so it should work automatically.

2. Webserver configuration with multiple projects' root

If you want to use gitweb with several project roots you can edit your Apache virtual host and gitweb configuration files in the following way.

The virtual host configuration (in Apache configuration file) should look like this:

```
<VirtualHost *:80>
    ServerName    git.example.org
    DocumentRoot  /pub/git
    SetEnv        GITWEB_CONFIG  /etc/gitweb.conf

    # turning on mod rewrite
    RewriteEngine on

    # make the front page an internal rewrite to the gitweb script
    RewriteRule ^/$  /cgi-bin/gitweb.cgi  [QSA,L,PT]

    # look for a public_git directory in unix users' home
    # http://git.example.org/~<user>/
    RewriteRule ^/\~([^\/]+)(/|/gitweb.cgi)?$   /cgi-bin/gitweb.cgi \
                [QSA,E=GITWEB_PROJECTROOT:/home/$1/public_git/,L,PT]

    # http://git.example.org/+<user>/
    #RewriteRule ^/\+([^\/]+)(/|/gitweb.cgi)?$  /cgi-bin/gitweb.cgi \
                 [QSA,E=GITWEB_PROJECTROOT:/home/$1/public_git/,L,PT]

    # http://git.example.org/user/<user>/
    #RewriteRule ^/user/([^\/]+)/(gitweb.cgi)?$ /cgi-bin/gitweb.cgi \
                 [QSA,E=GITWEB_PROJECTROOT:/home/$1/public_git/,L,PT]

    # defined list of project roots
    RewriteRule ^/scm(/|/gitweb.cgi)?$ /cgi-bin/gitweb.cgi \
                [QSA,E=GITWEB_PROJECTROOT:/pub/scm/,L,PT]
    RewriteRule ^/var(/|/gitweb.cgi)?$ /cgi-bin/gitweb.cgi \
                [QSA,E=GITWEB_PROJECTROOT:/var/git/,L,PT]

    # make access for "dumb clients" work
    RewriteRule ^/(.*\.git/(?!/?(HEAD|info|objects|refs)).*)?$ \
                /cgi-bin/gitweb.cgi%{REQUEST_URI}  [L,PT]
</VirtualHost>
```

Here actual project root is passed to gitweb via *GITWEB_PROJECT_ROOT* environment variable from a web server, so you need to put the following line in gitweb configuration file (*/etc/gitweb.conf* in above example):

```
$projectroot = $ENV{'GITWEB_PROJECTROOT'} || "/pub/git";
```

Note that this requires to be set for each request, so either *$per_request_config* must be false, or the above must be put in code referenced by *$per_request_config*;

These configurations enable two things. First, each unix user (*<user>*) of the server will be able to browse through gitweb Git repositories found in *~/public_git/* with the following url:

```
http://git.example.org/~<user>/
```

If you do not want this feature on your server just remove the second rewrite rule.

If you already use mod_userdir` in your virtual host or you don't want to use the '~ as first character, just comment or remove the second rewrite rule, and uncomment one of the following according to what you want.

Second, repositories found in */pub/scm/* and */var/git/* will be accessible through *http://git.example.org/scm/* and *http://git.example.org/var/*. You can add as many project roots as you want by adding rewrite rules like the third and the fourth.

3. PATH_INFO usage

If you enable PATH_INFO usage in gitweb by putting

```
$feature{'pathinfo'}{'default'} = [1];
```

in your gitweb configuration file, it is possible to set up your server so that it consumes and produces URLs in the form

```
http://git.example.com/project.git/shortlog/sometag
```

i.e. without *gitweb.cgi* part, by using a configuration such as the following. This configuration assumes that */var/www/gitweb* is the DocumentRoot of your webserver, contains the gitweb.cgi script and complementary static files (stylesheet, favicon, JavaScript):

```
<VirtualHost *:80>
        ServerAlias git.example.com

        DocumentRoot /var/www/gitweb

        <Directory /var/www/gitweb>
                Options ExecCGI
                AddHandler cgi-script cgi

                DirectoryIndex gitweb.cgi

                RewriteEngine On
                RewriteCond %{REQUEST_FILENAME} !-f
                RewriteCond %{REQUEST_FILENAME} !-d
                RewriteRule ^.* /gitweb.cgi/$0 [L,PT]
        </Directory>
</VirtualHost>
```

The rewrite rule guarantees that existing static files will be properly served, whereas any other URL will be passed to gitweb as PATH_INFO parameter.

Notice that in this case you don't need special settings for *@stylesheets*, *$my_uri* and *$home_link*, but you lose "dumb client" access to your project .git dirs (described in "Single URL for gitweb and for fetching" section). A possible workaround for the latter is the following: in your project root dir (e.g. */pub/git*) have the projects named **without** a .git extension (e.g. */pub/git/project* instead of */pub/git/project.git*) and configure Apache as follows:

```
<VirtualHost *:80>
        ServerAlias git.example.com

        DocumentRoot /var/www/gitweb

        AliasMatch ^(/.*?)(\.git)(/.*)?$ /pub/git$1$3
        <Directory /var/www/gitweb>
                Options ExecCGI
                AddHandler cgi-script cgi

                DirectoryIndex gitweb.cgi
```

```
                RewriteEngine On
                RewriteCond %{REQUEST_FILENAME} !-f
                RewriteCond %{REQUEST_FILENAME} !-d
                RewriteRule ^.* /gitweb.cgi/$0 [L,PT]
        </Directory>
</VirtualHost>
```

The additional AliasMatch makes it so that

```
http://git.example.com/project.git
```

will give raw access to the project's Git dir (so that the project can be cloned), while

```
http://git.example.com/project
```

will provide human-friendly gitweb access.

This solution is not 100% bulletproof, in the sense that if some project has a named ref (branch, tag) starting with *git/*, then paths such as

```
http://git.example.com/project/command/abranch..git/abranch
```

will fail with a 404 error.

BUGS

Please report any bugs or feature requests to git@vger.kernel.org [mailto:git@vger.kernel.org], putting "gitweb" in the subject of email.

SEE ALSO

Section G.4.17, “gitweb.conf(5)”, Section G.3.72, “git-instaweb(1)”

gitweb/README, *gitweb/INSTALL*

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.17. gitweb.conf(5)

2024-07-02

Revision History

2024-07-02

NAME

gitweb.conf - Gitweb (Git web interface) configuration file

SYNOPSIS

/etc/gitweb.conf, /etc/gitweb-common.conf, $GITWEBDIR/gitweb_config.perl

DESCRIPTION

The gitweb CGI script for viewing Git repositories over the web uses a perl script fragment as its configuration file. You can set variables using "*our $variable = value*"; text from a "#" character until the end of a line is ignored. See **perlsyn**(1) for details.

An example:

```
# gitweb configuration file for http://git.example.org
```

```
#
our $projectroot = "/srv/git"; # FHS recommendation
our $site_name = 'Example.org >> Repos';
```

The configuration file is used to override the default settings that were built into gitweb at the time the *gitweb.cgi* script was generated.

While one could just alter the configuration settings in the gitweb CGI itself, those changes would be lost upon upgrade. Configuration settings might also be placed into a file in the same directory as the CGI script with the default name *gitweb_config.perl* -- allowing one to have multiple gitweb instances with different configurations by the use of symlinks.

Note that some configuration can be controlled on per-repository rather than gitweb-wide basis: see "Per-repository gitweb configuration" subsection on Section G.4.16, “gitweb(1)” manpage.

DISCUSSION

Gitweb reads configuration data from the following sources in the following order:

- built-in values (some set during build stage),
- common system-wide configuration file (defaults to */etc/gitweb-common.conf*),
- either per-instance configuration file (defaults to *gitweb_config.perl* in the same directory as the installed gitweb), or if it does not exist then fallback system-wide configuration file (defaults to */etc/gitweb.conf*).

Values obtained in later configuration files override values obtained earlier in the above sequence.

Locations of the common system-wide configuration file, the fallback system-wide configuration file and the per-instance configuration file are defined at compile time using build-time Makefile configuration variables, respectively *GITWEB_CONFIG_COMMON*, *GITWEB_CONFIG_SYSTEM* and *GITWEB_CONFIG*.

You can also override locations of gitweb configuration files during runtime by setting the following environment variables: *GITWEB_CONFIG_COMMON*, *GITWEB_CONFIG_SYSTEM* and *GITWEB_CONFIG* to a non-empty value.

The syntax of the configuration files is that of Perl, since these files are handled by sourcing them as fragments of Perl code (the language that gitweb itself is written in). Variables are typically set using the *our* qualifier (as in "*our $variable = <value>;*") to avoid syntax errors if a new version of gitweb no longer uses a variable and therefore stops declaring it.

You can include other configuration file using read_config_file() subroutine. For example, one might want to put gitweb configuration related to access control for viewing repositories via Gitolite (one of Git repository management tools) in a separate file, e.g. in */etc/gitweb-gitolite.conf*. To include it, put

```
read_config_file("/etc/gitweb-gitolite.conf");
```

somewhere in gitweb configuration file used, e.g. in per-installation gitweb configuration file. Note that read_config_file() checks itself that the file it reads exists, and does nothing if it is not found. It also handles errors in included file.

The default configuration with no configuration file at all may work perfectly well for some installations. Still, a configuration file is useful for customizing or tweaking the behavior of gitweb in many ways, and some optional features will not be present unless explicitly enabled using the configurable *%features* variable (see also "Configuring gitweb features" section below).

CONFIGURATION VARIABLES

Some configuration variables have their default values (embedded in the CGI script) set during building gitweb -- if that is the case, this fact is put in their description. See gitweb's *INSTALL* file for instructions on building and installing gitweb.

1. Location of repositories

The configuration variables described below control how gitweb finds Git repositories, and how repositories are displayed and accessed.

See also "Repositories" and later subsections in Section G.4.16, “gitweb(1)” manpage.

$projectroot

Absolute filesystem path which will be prepended to project path; the path to repository is *$projectroot/$project*. Set to *$GITWEB_PROJECTROOT* during installation. This variable has to be set correctly for gitweb to find repositories.

For example, if *$projectroot* is set to "/srv/git" by putting the following in gitweb config file:

```
our $projectroot = "/srv/git";
```

then

```
http://git.example.com/gitweb.cgi?p=foo/bar.git
```

and its path_info based equivalent

```
http://git.example.com/gitweb.cgi/foo/bar.git
```

will map to the path */srv/git/foo/bar.git* on the filesystem.

$projects_list

Name of a plain text file listing projects, or a name of directory to be scanned for projects.

Project list files should list one project per line, with each line having the following format

```
<URI-encoded filesystem path to repository> SP <URI-encoded repository
 owner>
```

The default value of this variable is determined by the *GITWEB_LIST* makefile variable at installation time. If this variable is empty, gitweb will fall back to scanning the *$projectroot* directory for repositories.

$project_maxdepth

If *$projects_list* variable is unset, gitweb will recursively scan filesystem for Git repositories. The *$project_maxdepth* is used to limit traversing depth, relative to *$projectroot* (starting point); it means that directories which are further from *$projectroot* than *$project_maxdepth* will be skipped.

It is purely performance optimization, originally intended for MacOS X, where recursive directory traversal is slow. Gitweb follows symbolic links, but it detects cycles, ignoring any duplicate files and directories.

The default value of this variable is determined by the build-time configuration variable *GITWEB_PROJECT_MAXDEPTH*, which defaults to 2007.

$export_ok

Show repository only if this file exists (in repository). Only effective if this variable evaluates to true. Can be set when building gitweb by setting *GITWEB_EXPORT_OK*. This path is relative to *GIT_DIR*. git-daemon[1] uses *git-daemon-export-ok*, unless started with *--export-all*. By default this variable is not set, which means that this feature is turned off.

$export_auth_hook

Function used to determine which repositories should be shown. This subroutine should take one parameter, the full path to a project, and if it returns true, that project will be included in the projects list and can be accessed through gitweb as long as it fulfills the other requirements described by $export_ok, $projects_list, and $projects_maxdepth. Example:

```
our $export_auth_hook = sub { return -e "$_[0]/git-daemon-export-ok"; };
```

though the above might be done by using *$export_ok* instead

```
our $export_ok = "git-daemon-export-ok";
```

If not set (default), it means that this feature is disabled.

See also more involved example in "Controlling access to Git repositories" subsection on Section G.4.16, “gitweb(1)” manpage.

$strict_export

Only allow viewing of repositories also shown on the overview page. This for example makes *$export_ok* file decide if repository is available and not only if it is shown. If *$projects_list* points to file with list of project, only those repositories listed would be available for gitweb. Can be set during building gitweb via *GITWEB_STRICT_EXPORT*. By default this variable is not set, which means that you can directly access those repositories that are hidden from projects list page (e.g. the are not listed in the $projects_list file).

2. Finding files

The following configuration variables tell gitweb where to find files. The values of these variables are paths on the filesystem.

$GIT

Core git executable to use. By default set to *$GIT_BINDIR/git*, which in turn is by default set to *$(bindir)/git*. If you use Git installed from a binary package, you should usually set this to "/usr/bin/git". This can just be "git" if your web server has a sensible PATH; from security point of view it is better to use absolute path to git binary. If you have multiple Git versions installed it can be used to choose which one to use. Must be (correctly) set for gitweb to be able to work.

$mimetypes_file

File to use for (filename extension based) guessing of MIME types before trying */etc/mime.types*. **NOTE** that this path, if relative, is taken as relative to the current Git repository, not to CGI script. If unset, only */etc/mime.types* is used (if present on filesystem). If no mimetypes file is found, mimetype guessing based on extension of file is disabled. Unset by default.

$highlight_bin

Path to the highlight executable to use (it must be the one from http://andre-simon.de/zip/download.php due to assumptions about parameters and output). By default set to *highlight*; set it to full path to highlight executable if it is not installed on your web server's PATH. Note that *highlight* feature must be set for gitweb to actually use syntax highlighting.

NOTE: for a file to be highlighted, its syntax type must be detected and that syntax must be supported by "highlight". The default syntax detection is minimal, and there are many supported syntax types with no detection by default. There are three options for adding syntax detection. The first and second priority are *%highlight_basename* and *%highlight_ext*, which detect based on basename (the full filename, for example "Makefile") and extension (for example "sh"). The keys of these hashes are the basename and extension, respectively, and the value for a given key is the name of the syntax to be passed via *--syntax <syntax>* to "highlight". The last priority is the "highlight" configuration of *Shebang* regular expressions to detect the language based on the first line in the file, (for example, matching the line "#!/bin/bash"). See the highlight documentation and the default config at /etc/highlight/filetypes.conf for more details.

For example if repositories you are hosting use "phtml" extension for PHP files, and you want to have correct syntax-highlighting for those files, you can add the following to gitweb configuration:

```
our %highlight_ext;
$highlight_ext{'phtml'} = 'php';
```

3. Links and their targets

The configuration variables described below configure some of gitweb links: their target and their look (text or image), and where to find page prerequisites (stylesheet, favicon, images, scripts). Usually they are left at their default values, with the possible exception of *@stylesheets* variable.

@stylesheets

List of URIs of stylesheets (relative to the base URI of a page). You might specify more than one stylesheet, for example to use "gitweb.css" as base with site specific modifications in a separate stylesheet to make it easier to upgrade gitweb. For example, you can add a *site* stylesheet by putting

```
push @stylesheets, "gitweb-site.css";
```

in the gitweb config file. Those values that are relative paths are relative to base URI of gitweb.

This list should contain the URI of gitweb's standard stylesheet. The default URI of gitweb stylesheet can be set at build time using the *GITWEB_CSS* makefile variable. Its default value is *static/gitweb.css* (or *static/gitweb.min.css* if the *CSSMIN* variable is defined, i.e. if CSS minifier is used during build).

Note: there is also a legacy *$stylesheet* configuration variable, which was used by older gitweb. If *$stylesheet* variable is defined, only CSS stylesheet given by this variable is used by gitweb.

$logo

Points to the location where you put *git-logo.png* on your web server, or to be more the generic URI of logo, 72x27 size). This image is displayed in the top right corner of each gitweb page and used as a logo for the Atom feed. Relative to the base URI of gitweb (as a path). Can be adjusted when building gitweb using *GITWEB_LOGO* variable By default set to *static/git-logo.png*.

$favicon

Points to the location where you put *git-favicon.png* on your web server, or to be more the generic URI of favicon, which will be served as "image/png" type. Web browsers that support favicons (website icons) may display them in the browser's URL bar and next to the site name in bookmarks. Relative to the base URI of gitweb. Can be adjusted at build time using *GITWEB_FAVICON* variable. By default set to *static/git-favicon.png*.

$javascript

Points to the location where you put *gitweb.js* on your web server, or to be more generic the URI of JavaScript code used by gitweb. Relative to the base URI of gitweb. Can be set at build time using the *GITWEB_JS* build-time configuration variable.

The default value is either *static/gitweb.js*, or *static/gitweb.min.js* if the *JSMIN* build variable was defined, i.e. if JavaScript minifier was used at build time. **Note** that this single file is generated from multiple individual JavaScript "modules".

$home_link

Target of the home link on the top of all pages (the first part of view "breadcrumbs"). By default it is set to the absolute URI of a current page (to the value of *$my_uri* variable, or to "/" if *$my_uri* is undefined or is an empty string).

$home_link_str

Label for the "home link" at the top of all pages, leading to *$home_link* (usually the main gitweb page, which contains the projects list). It is used as the first component of gitweb's "breadcrumb trail": *<home-link> / <project> / <action>*. Can be set at build time using the *GITWEB_HOME_LINK_STR* variable. By default it is set to "projects", as this link leads to the list of projects. Another popular choice is to set it to the name of site. Note that it is treated as raw HTML so it should not be set from untrusted sources.

@extra_breadcrumbs

Additional links to be added to the start of the breadcrumb trail before the home link, to pages that are logically "above" the gitweb projects list, such as the organization and department which host the gitweb server. Each element of the list is a reference to an array, in which element 0 is the link text (equivalent to *$home_link_str*) and element 1 is the target URL (equivalent to *$home_link*).

For example, the following setting produces a breadcrumb trail like "home / dev / projects / …" where "projects" is the home link.

```
our @extra_breadcrumbs = (
  [ 'home' => 'https://www.example.org/' ],
  [ 'dev'  => 'https://dev.example.org/' ],
);
```

$logo_url , $logo_label

URI and label (title) for the Git logo link (or your site logo, if you chose to use different logo image). By default, these both refer to Git homepage, https://git-scm.com; in the past, they pointed to Git documentation at https://www.kernel.org.

4. Changing gitweb's look

You can adjust how pages generated by gitweb look using the variables described below. You can change the site name, add common headers and footers for all pages, and add a description of this gitweb installation on its main page (which is the projects list page), etc.

$site_name

Name of your site or organization, to appear in page titles. Set it to something descriptive for clearer bookmarks etc. If this variable is not set or is, then gitweb uses the value of the *SERVER_NAME CGI* environment variable, setting site name to "$SERVER_NAME Git", or "Untitled Git" if this variable is not set (e.g. if running gitweb as standalone script).

Can be set using the *GITWEB_SITENAME* at build time. Unset by default.

$site_html_head_string

HTML snippet to be included in the <head> section of each page. Can be set using *GITWEB_SITE_HTML_HEAD_STRING* at build time. No default value.

$site_header

Name of a file with HTML to be included at the top of each page. Relative to the directory containing the *gitweb.cgi* script. Can be set using *GITWEB_SITE_HEADER* at build time. No default value.

$site_footer

Name of a file with HTML to be included at the bottom of each page. Relative to the directory containing the *gitweb.cgi* script. Can be set using *GITWEB_SITE_FOOTER* at build time. No default value.

$home_text

Name of a HTML file which, if it exists, is included on the gitweb projects overview page ("projects_list" view). Relative to the directory containing the gitweb.cgi script. Default value can be adjusted during build time using *GITWEB_HOMETEXT* variable. By default set to *indextext.html*.

$projects_list_description_width

The width (in characters) of the "Description" column of the projects list. Longer descriptions will be truncated (trying to cut at word boundary); the full description is available in the *title* attribute (usually shown on mouseover). The default is 25, which might be too small if you use long project descriptions.

$default_projects_order

Default value of ordering of projects on projects list page, which means the ordering used if you don't explicitly sort projects list (if there is no "o" CGI query parameter in the URL). Valid values are "none" (unsorted), "project" (projects are by project name, i.e. path to repository relative to *$projectroot*), "descr" (project description), "owner", and "age" (by date of most current commit).

Default value is "project". Unknown value means unsorted.

5. Changing gitweb's behavior

These configuration variables control *internal* gitweb behavior.

$default_blob_plain_mimetype

Default mimetype for the blob_plain (raw) view, if mimetype checking doesn't result in some other type; by default "text/plain". Gitweb guesses mimetype of a file to display based on extension of its filename, using *$mimetypes_file* (if set and file exists) and */etc/mime.types* files (see **mime.types**(5) manpage; only filename extension rules are supported by gitweb).

$default_text_plain_charset

Default charset for text files. If this is not set, the web server configuration will be used. Unset by default.

$fallback_encoding

Gitweb assumes this charset when a line contains non-UTF-8 characters. The fallback decoding is used without error checking, so it can be even "utf-8". The value must be a valid encoding; see the **Encoding::Supported**(3pm) man page for a list. The default is "latin1", aka. "iso-8859-1".

@diff_opts

Rename detection options for git-diff and git-diff-tree. The default is ('-M'); set it to ('-C') or ('-C', '-C') to also detect copies, or set it to () i.e. empty list if you don't want to have renames detection.

Note that rename and especially copy detection can be quite CPU-intensive. Note also that non Git tools can have problems with patches generated with options mentioned above, especially when they involve file copies ('-C') or criss-cross renames ('-B').

6. Some optional features and policies

Most of features are configured via *%feature* hash; however some of extra gitweb features can be turned on and configured using variables described below. This list beside configuration variables that control how gitweb looks does contain variables configuring administrative side of gitweb (e.g. cross-site scripting prevention; admittedly this as side effect affects how "summary" pages look like, or load limiting).

@git_base_url_list

List of Git base URLs. These URLs are used to generate URLs describing from where to fetch a project, which are shown on project summary page. The full fetch URL is "*$git_base_url/$project*", for each element of this list. You can set up multiple base URLs (for example one for *git://* protocol, and one for *http://* protocol).

Note that per repository configuration can be set in *$GIT_DIR/cloneurl* file, or as values of multi-value *gitweb.url* configuration variable in project config. Per-repository configuration takes precedence over value composed from *@git_base_url_list* elements and project name.

You can setup one single value (single entry/item in this list) at build time by setting the *GITWEB_BASE_URL* build-time configuration variable. By default it is set to (), i.e. an empty list. This means that gitweb would not try to create project URL (to fetch) from project name.

$projects_list_group_categories

Whether to enable the grouping of projects by category on the project list page. The category of a project is determined by the *$GIT_DIR/category* file or the *gitweb.category* variable in each repository's configuration. Disabled by default (set to 0).

$project_list_default_category

Default category for projects for which none is specified. If this is set to the empty string, such projects will remain uncategorized and listed at the top, above categorized projects. Used only if project categories are enabled, which means if *$projects_list_group_categories* is true. By default set to "" (empty string).

$prevent_xss

If true, some gitweb features are disabled to prevent content in repositories from launching cross-site scripting (XSS) attacks. Set this to true if you don't trust the content of your repositories. False by default (set to 0).

$maxload

Used to set the maximum load that we will still respond to gitweb queries. If the server load exceeds this value then gitweb will return "503 Service Unavailable" error. The server load is taken to be 0 if gitweb cannot determine its value. Currently it works only on Linux, where it uses */proc/loadavg*; the load there is the number of active tasks on the system -- processes that are actually running -- averaged over the last minute.

Set *$maxload* to undefined value (*undef*) to turn this feature off. The default value is 300.

$omit_age_column

If true, omit the column with date of the most current commit on the projects list page. It can save a bit of I/O and a fork per repository.

$omit_owner

If true prevents displaying information about repository owner.

$per_request_config

If this is set to code reference, it will be run once for each request. You can set parts of configuration that change per session this way. For example, one might use the following code in a gitweb configuration file

```
our $per_request_config = sub {
        $ENV{GL_USER} = $cgi->remote_user || "gitweb";
};
```

If *$per_request_config* is not a code reference, it is interpreted as boolean value. If it is true gitweb will process config files once per request, and if it is false gitweb will process config files only once, each time it is executed. True by default (set to 1).

NOTE: *$my_url*, *$my_uri*, and *$base_url* are overwritten with their default values before every request, so if you want to change them, be sure to set this variable to true or a code reference effecting the desired changes.

This variable matters only when using persistent web environments that serve multiple requests using single gitweb instance, like mod_perl, FastCGI or Plackup.

7. Other variables

Usually you should not need to change (adjust) any of configuration variables described below; they should be automatically set by gitweb to correct value.

$version

Gitweb version, set automatically when creating gitweb.cgi from gitweb.perl. You might want to modify it if you are running modified gitweb, for example

```
our $version .= " with caching";
```

if you run modified version of gitweb with caching support. This variable is purely informational, used e.g. in the "generator" meta header in HTML header.

$my_url , $my_uri

Full URL and absolute URL of the gitweb script; in earlier versions of gitweb you might have need to set those variables, but now there should be no need to do it. See *$per_request_config* if you need to set them still.

$base_url

Base URL for relative URLs in pages generated by gitweb, (e.g. *$logo*, *$favicon*, *@stylesheets* if they are relative URLs), needed and used *<base href="$base_url">* only for URLs with nonempty PATH_INFO. Usually gitweb sets its value correctly, and there is no need to set this variable, e.g. to $my_uri or "/". See *$per_request_config* if you need to override it anyway.

CONFIGURING GITWEB FEATURES

Many gitweb features can be enabled (or disabled) and configured using the *%feature* hash. Names of gitweb features are keys of this hash.

Each *%feature* hash element is a hash reference and has the following structure:

```
"<feature-name>" => {
        "sub" => <feature-sub-(subroutine)>,
        "override" => <allow-override-(boolean)>,
        "default" => [ <options>... ]
},
```

Some features cannot be overridden per project. For those features the structure of appropriate *%feature* hash element has a simpler form:

```
"<feature-name>" => {
        "override" => 0,
        "default" => [ <options>... ]
},
```

As one can see it lacks the 'sub' element.

The meaning of each part of feature configuration is described below:

default

List (array reference) of feature parameters (if there are any), used also to toggle (enable or disable) given feature.

Note that it is currently **always** an array reference, even if feature doesn't accept any configuration parameters, and 'default' is used only to turn it on or off. In such case you turn feature on by setting this element to *[1]*, and torn it off by setting it to *[0]*. See also the passage about the "blame" feature in the "Examples" section.

To disable features that accept parameters (are configurable), you need to set this element to empty list i.e. *[]*.

override

If this field has a true value then the given feature is overridable, which means that it can be configured (or enabled/disabled) on a per-repository basis.

Usually given "<feature>" is configurable via the *gitweb.<feature>* config variable in the per-repository Git configuration file.

Note that no feature is overridable by default.

sub

Internal detail of implementation. What is important is that if this field is not present then per-repository override for given feature is not supported.

You wouldn't need to ever change it in gitweb config file.

1. Features in *%feature*

The gitweb features that are configurable via *%feature* hash are listed below. This should be a complete list, but ultimately the authoritative and complete list is in gitweb.cgi source code, with features described in the comments.

blame

Enable the "blame" and "blame_incremental" blob views, showing for each line the last commit that modified it; see Section G.3.9, “git-blame(1)”. This can be very CPU-intensive and is therefore disabled by default.

This feature can be configured on a per-repository basis via repository's *gitweb.blame* configuration variable (boolean).

snapshot

Enable and configure the "snapshot" action, which allows user to download a compressed archive of any tree or commit, as produced by Section G.3.7, “git-archive(1)” and possibly additionally compressed. This can potentially generate high traffic if you have large project.

The value of 'default' is a list of names of snapshot formats, defined in *%known_snapshot_formats* hash, that you wish to offer. Supported formats include "tgz", "tbz2", "txz" (gzip/bzip2/xz compressed tar archive) and "zip"; please consult gitweb sources for a definitive list. By default only "tgz" is offered.

This feature can be configured on a per-repository basis via repository's *gitweb.snapshot* configuration variable, which contains a comma separated list of formats or "none" to disable snapshots. Unknown values are ignored.

grep

Enable grep search, which lists the files in currently selected tree (directory) containing the given string; see Section G.3.60, “git-grep(1)”. This can be potentially CPU-intensive, of course. Enabled by default.

This feature can be configured on a per-repository basis via repository's *gitweb.grep* configuration variable (boolean).

pickaxe

Enable the so called pickaxe search, which will list the commits that introduced or removed a given string in a file. This can be practical and quite faster alternative to "blame" action, but it is still potentially CPU-intensive. Enabled by default.

The pickaxe search is described in Section G.3.74, “git-log(1)” (the description of *-S<string>* option, which refers to pickaxe entry in Section G.4.4, “gitdiffcore(7)” for more details).

This feature can be configured on a per-repository basis by setting repository's *gitweb.pickaxe* configuration variable (boolean).

show-sizes

Enable showing size of blobs (ordinary files) in a "tree" view, in a separate column, similar to what *ls -l* does; see description of *-l* option in Section G.3.77, “git-ls-tree(1)” manpage. This costs a bit of I/O. Enabled by default.

This feature can be configured on a per-repository basis via repository's *gitweb.showSizes* configuration variable (boolean).

patches

Enable and configure "patches" view, which displays list of commits in email (plain text) output format; see also Section G.3.54, “git-format-patch(1)”. The value is the maximum number of patches in a patchset generated in "patches" view. Set the *default* field to a list containing single item of or to an empty list to disable patch view, or to a list containing a single negative number to remove any limit. Default value is 16.

This feature can be configured on a per-repository basis via repository's *gitweb.patches* configuration variable (integer).

avatar

Avatar support. When this feature is enabled, views such as "shortlog" or "commit" will display an avatar associated with the email of each committer and author.

Currently available providers are **"gravatar"** and **"picon"**. Only one provider at a time can be selected (*default* is one element list). If an unknown provider is specified, the feature is disabled. **Note** that some providers might require extra Perl packages to be installed; see *gitweb/INSTALL* for more details.

This feature can be configured on a per-repository basis via repository's *gitweb.avatar* configuration variable.

See also *%avatar_size* with pixel sizes for icons and avatars ("default" is used for one-line like "log" and "shortlog", "double" is used for two-line like "commit", "commitdiff" or "tag"). If the default font sizes or lineheights are changed (e.g. via adding extra CSS stylesheet in *@stylesheets*), it may be appropriate to change these values.

email-privacy

Redact e-mail addresses from the generated HTML, etc. content. This obscures e-mail addresses retrieved from the author/committer and comment sections of the Git log. It is meant to hinder web crawlers that harvest and abuse addresses. Such crawlers may not respect robots.txt. Note that users and user tools also see the addresses as redacted. If Gitweb is not the final step in a workflow then subsequent steps may misbehave because of the redacted information they receive. Disabled by default.

highlight

Server-side syntax highlight support in "blob" view. It requires *$highlight_bin* program to be available (see the description of this variable in the "Configuration variables" section above), and therefore is disabled by default.

This feature can be configured on a per-repository basis via repository's *gitweb.highlight* configuration variable (boolean).

remote_heads

Enable displaying remote heads (remote-tracking branches) in the "heads" list. In most cases the list of remote-tracking branches is an unnecessary internal private detail, and this feature is therefore disabled by default. Section G.3.72, “git-instaweb(1)”, which is usually used to browse local repositories, enables and uses this feature.

This feature can be configured on a per-repository basis via repository's *gitweb.remote_heads* configuration variable (boolean).

The remaining features cannot be overridden on a per project basis.

search

Enable text search, which will list the commits which match author, committer or commit text to a given string; see the description of *--author*, *--committer* and *--grep* options in Section G.3.74, “git-log(1)” manpage. Enabled by default.

Project specific override is not supported.

forks

If this feature is enabled, gitweb considers projects in subdirectories of project root (basename) to be forks of existing projects. For each project `$projname.git`, projects in the `$projname/` directory and its subdirectories will not be shown in the main projects list. Instead, a '+' mark is shown next to *$projname*, which links to a "forks" view that lists all the forks (all projects in *$projname/* subdirectory). Additionally a "forks" view for a project is linked from project summary page.

If the project list is taken from a file (`$projects_list` points to a file), forks are only recognized if they are listed after the main project in that file.

Project specific override is not supported.

actions

Insert custom links to the action bar of all project pages. This allows you to link to third-party scripts integrating into gitweb.

The "default" value consists of a list of triplets in the form ("<label>", "<link>", "<position>")` where "position" is the label after which to insert the link, "link" is a format string where *%n* expands to the project name, *%f* to the project path within the filesystem (i.e. "$projectroot/$project"), *%h* to the current hash ('h gitweb parameter) and %b` to the current hash base ('hb gitweb parameter); %%` expands to '%.

For example, at the time this page was written, the https://repo.or.cz Git hosting site set it to the following to enable graphical log (using the third party tool **git-browser**):

```
$feature{'actions'}{'default'} =
        [ ('graphiclog', '/git-browser/by-commit.html?r=%n',
 'summary')];
```

This adds a link titled "graphiclog" after the "summary" link, leading to *git-browser* script, passing *r=<project>* as a query parameter.

Project specific override is not supported.

timed

Enable displaying how much time and how many Git commands it took to generate and display each page in the page footer (at the bottom of page). For example the footer might contain: "This page took 6.53325 seconds and 13 Git commands to generate." Disabled by default.

Project specific override is not supported.

javascript-timezone

Enable and configure the ability to change a common time zone for dates in gitweb output via JavaScript. Dates in gitweb output include authordate and committerdate in "commit", "commitdiff" and "log" views, and taggerdate in "tag" view. Enabled by default.

The value is a list of three values: a default time zone (for if the client hasn't selected some other time zone and saved it in a cookie), a name of cookie where to store selected time zone, and a CSS class used to mark up dates for manipulation. If you want to turn this feature off, set "default" to empty list: *[]*.

Typical gitweb config files will only change starting (default) time zone, and leave other elements at their default values:

```
$feature{'javascript-timezone'}{'default'}[0] = "utc";
```

The example configuration presented here is guaranteed to be backwards and forward compatible.

Time zone values can be "local" (for local time zone that browser uses), "utc" (what gitweb uses when JavaScript or this feature is disabled), or numerical time zones in the form of "+/-HHMM", such as "+0200".

Project specific override is not supported.

extra-branch-refs

List of additional directories under "refs" which are going to be used as branch refs. For example if you have a gerrit setup where all branches under refs/heads/ are official, push-after-review ones and branches under refs/sandbox/, refs/wip and refs/other are user ones where permissions are much wider, then you might want to set this variable as follows:

```
$feature{'extra-branch-refs'}{'default'} =
        ['sandbox', 'wip', 'other'];
```

This feature can be configured on per-repository basis after setting $feature{*extra-branch-refs*}{*override*} to true, via repository's *gitweb.extraBranchRefs* configuration variable, which contains a space separated list of refs. An example:

```
[gitweb]
        extraBranchRefs = sandbox wip other
```

The gitweb.extraBranchRefs is actually a multi-valued configuration variable, so following example is also correct and the result is the same as of the snippet above:

```
[gitweb]
        extraBranchRefs = sandbox
        extraBranchRefs = wip other
```

It is an error to specify a ref that does not pass "git check-ref-format" scrutiny. Duplicated values are filtered.

EXAMPLES

To enable blame, pickaxe search, and snapshot support (allowing "tar.gz" and "zip" snapshots), while allowing individual projects to turn them off, put the following in your GITWEB_CONFIG file:

```
$feature{'blame'}{'default'} = [1];
$feature{'blame'}{'override'} = 1;

$feature{'pickaxe'}{'default'} = [1];
$feature{'pickaxe'}{'override'} = 1;

$feature{'snapshot'}{'default'} = ['zip', 'tgz'];
$feature{'snapshot'}{'override'} = 1;
```

If you allow overriding for the snapshot feature, you can specify which snapshot formats are globally disabled. You can also add any command-line options you want (such as setting the compression level). For instance, you can disable Zip compressed snapshots and set **gzip**(1) to run at level 6 by adding the following lines to your gitweb configuration file:

```
$known_snapshot_formats{'zip'}{'disabled'} = 1;
$known_snapshot_formats{'tgz'}{'compressor'} = ['gzip','-6'];
```

BUGS

Debugging would be easier if the fallback configuration file (*/etc/gitweb.conf*) and environment variable to override its location (*GITWEB_CONFIG_SYSTEM*) had names reflecting their "fallback" role. The current names are kept to avoid breaking working setups.

ENVIRONMENT

The location of per-instance and system-wide configuration files can be overridden using the following environment variables:

GITWEB_CONFIG

Sets location of per-instance configuration file.

GITWEB_CONFIG_SYSTEM

Sets location of fallback system-wide configuration file. This file is read only if per-instance one does not exist.

GITWEB_CONFIG_COMMON

Sets location of common system-wide configuration file.

FILES

gitweb_config.perl

This is default name of per-instance configuration file. The format of this file is described above.

/etc/gitweb.conf

This is default name of fallback system-wide configuration file. This file is used only if per-instance configuration variable is not found.

/etc/gitweb-common.conf

This is default name of common system-wide configuration file.

SEE ALSO

Section G.4.16, “gitweb(1)”, Section G.3.72, “git-instaweb(1)”

gitweb/README, *gitweb/INSTALL*

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.18. gitworkflows(7)

2024-07-02
Revision History
2024-07-02

NAME

gitworkflows - An overview of recommended workflows with Git

SYNOPSIS

git *

DESCRIPTION

This document attempts to write down and motivate some of the workflow elements used for *git.git* itself. Many ideas apply in general, though the full workflow is rarely required for smaller projects with fewer people involved.

We formulate a set of *rules* for quick reference, while the prose tries to motivate each of them. Do not always take them literally; you should value good reasons for your actions higher than manpages such as this one.

SEPARATE CHANGES

As a general rule, you should try to split your changes into small logical steps, and commit each of them. They should be consistent, working independently of any later commits, pass the test suite, etc. This makes the review

process much easier, and the history much more useful for later inspection and analysis, for example with Section G.3.9, “git-blame(1)” and Section G.3.8, “git-bisect(1)”.

To achieve this, try to split your work into small steps from the very beginning. It is always easier to squash a few commits together than to split one big commit into several. Don't be afraid of making too small or imperfect steps along the way. You can always go back later and edit the commits with *git rebase --interactive* before you publish them. You can use *git stash push --keep-index* to run the test suite independent of other uncommitted changes; see the EXAMPLES section of Section G.3.137, “git-stash(1)”.

MANAGING BRANCHES

There are two main tools that can be used to include changes from one branch on another: Section G.3.86, “git-merge(1)” and Section G.3.20, “git-cherry-pick(1)”.

Merges have many advantages, so we try to solve as many problems as possible with merges alone. Cherry-picking is still occasionally useful; see "Merging upwards" below for an example.

Most importantly, merging works at the branch level, while cherry-picking works at the commit level. This means that a merge can carry over the changes from 1, 10, or 1000 commits with equal ease, which in turn means the workflow scales much better to a large number of contributors (and contributions). Merges are also easier to understand because a merge commit is a "promise" that all changes from all its parents are now included.

There is a tradeoff of course: merges require a more careful branch management. The following subsections discuss the important points.

1. Graduation

As a given feature goes from experimental to stable, it also "graduates" between the corresponding branches of the software. *git.git* uses the following *integration branches*:

- *maint* tracks the commits that should go into the next "maintenance release", i.e., update of the last released stable version;
- *master* tracks the commits that should go into the next release;
- *next* is intended as a testing branch for topics being tested for stability for master.

There is a fourth official branch that is used slightly differently:

- *seen* (patches seen by the maintainer) is an integration branch for things that are not quite ready for inclusion yet (see "Integration Branches" below).

Each of the four branches is usually a direct descendant of the one above it.

Conceptually, the feature enters at an unstable branch (usually *next* or *seen*), and "graduates" to *master* for the next release once it is considered stable enough.

2. Merging upwards

The "downwards graduation" discussed above cannot be done by actually merging downwards, however, since that would merge *all* changes on the unstable branch into the stable one. Hence the following:

Always commit your fixes to the oldest supported branch that requires them. Then (periodically) merge the integration branches upwards into each other.

Example G.1. Merge upwards

This gives a very controlled flow of fixes. If you notice that you have applied a fix to e.g. *master* that is also required in *maint*, you will need to cherry-pick it (using Section G.3.20, “git-cherry-pick(1)”) downwards. This will happen a few times and is nothing to worry about unless you do it very frequently.

3. Topic branches

Any nontrivial feature will require several patches to implement, and may get extra bugfixes or improvements during its lifetime.

Committing everything directly on the integration branches leads to many problems: Bad commits cannot be undone, so they must be reverted one by one, which creates confusing histories and further error potential when you forget to revert part of a group of changes. Working in parallel mixes up the changes, creating further confusion.

Use of "topic branches" solves these problems. The name is pretty self explanatory, with a caveat that comes from the "merge upwards" rule above:

Make a side branch for every topic (feature, bugfix, …). Fork it off at the oldest integration branch that you will eventually want to merge it into.

Example G.2. Topic branches

Many things can then be done very naturally:

- To get the feature/bugfix into an integration branch, simply merge it. If the topic has evolved further in the meantime, merge again. (Note that you do not necessarily have to merge it to the oldest integration branch first. For example, you can first merge a bugfix to *next*, give it some testing time, and merge to *maint* when you know it is stable.)
- If you find you need new features from the branch *other* to continue working on your topic, merge *other* to *topic*. (However, do not do this "just habitually", see below.)
- If you find you forked off the wrong branch and want to move it "back in time", use Section G.3.107, “git-rebase(1)”.

Note that the last point clashes with the other two: a topic that has been merged elsewhere should not be rebased. See the section on RECOVERING FROM UPSTREAM REBASE in Section G.3.107, “git-rebase(1)”.

We should point out that "habitually" (regularly for no real reason) merging an integration branch into your topics -- and by extension, merging anything upstream into anything downstream on a regular basis -- is frowned upon:

Do not merge to downstream except with a good reason: upstream API changes affect your branch; your branch no longer merges to upstream cleanly; etc.

Example G.3. Merge to downstream only at well-defined points

Otherwise, the topic that was merged to suddenly contains more than a single (well-separated) change. The many resulting small merges will greatly clutter up history. Anyone who later investigates the history of a file will have to find out whether that merge affected the topic in development. An upstream might even inadvertently be merged into a "more stable" branch. And so on.

4. Throw-away integration

If you followed the last paragraph, you will now have many small topic branches, and occasionally wonder how they interact. Perhaps the result of merging them does not even work? But on the other hand, we want to avoid merging them anywhere "stable" because such merges cannot easily be undone.

The solution, of course, is to make a merge that we can undo: merge into a throw-away branch.

To test the interaction of several topics, merge them into a throw-away branch. You must never base any work on such a branch!

Example G.4. Throw-away integration branches

If you make it (very) clear that this branch is going to be deleted right after the testing, you can even publish this branch, for example to give the testers a chance to work with it, or other developers a chance to see if their in-progress work will be compatible. *git.git* has such an official throw-away integration branch called *seen*.

5. Branch management for a release

Assuming you are using the merge approach discussed above, when you are releasing your project you will need to do some additional branch management work.

A feature release is created from the *master* branch, since *master* tracks the commits that should go into the next feature release.

The *master* branch is supposed to be a superset of *maint*. If this condition does not hold, then *maint* contains some commits that are not included on *master*. The fixes represented by those commits will therefore not be included in your feature release.

To verify that *master* is indeed a superset of *maint*, use git log:

git log master..maint

Example G.5. Verify *master* is a superset of *maint*

This command should not list any commits. Otherwise, check out *master* and merge *maint* into it.

Now you can proceed with the creation of the feature release. Apply a tag to the tip of *master* indicating the release version:

git tag -s -m "Git X.Y.Z" vX.Y.Z master

Example G.6. Release tagging

You need to push the new tag to a public Git server (see "DISTRIBUTED WORKFLOWS" below). This makes the tag available to others tracking your project. The push could also trigger a post-update hook to perform release-related items such as building release tarballs and preformatted documentation pages.

Similarly, for a maintenance release, *maint* is tracking the commits to be released. Therefore, in the steps above simply tag and push *maint* rather than *master*.

6. Maintenance branch management after a feature release

After a feature release, you need to manage your maintenance branches.

First, if you wish to continue to release maintenance fixes for the feature release made before the recent one, then you must create another branch to track commits for that previous release.

To do this, the current maintenance branch is copied to another branch named with the previous release version number (e.g. maint-X.Y.(Z-1) where X.Y.Z is the current release).

git branch maint-X.Y.(Z-1) maint

Example G.7. Copy maint

The *maint* branch should now be fast-forwarded to the newly released code so that maintenance fixes can be tracked for the current release:

- *git checkout maint*
- *git merge --ff-only master*

Example G.8. Update maint to new release

If the merge fails because it is not a fast-forward, then it is possible some fixes on *maint* were missed in the feature release. This will not happen if the content of the branches was verified as described in the previous section.

7. Branch management for next and seen after a feature release

After a feature release, the integration branch *next* may optionally be rewound and rebuilt from the tip of *master* using the surviving topics on *next*:

- *git switch -C next master*
- *git merge ai/topic_in_next1*
- *git merge ai/topic_in_next2*
- …

Example G.9. Rewind and rebuild next

The advantage of doing this is that the history of *next* will be clean. For example, some topics merged into *next* may have initially looked promising, but were later found to be undesirable or premature. In such a case, the topic is reverted out of *next* but the fact remains in the history that it was once merged and reverted. By recreating *next*, you give another incarnation of such topics a clean slate to retry, and a feature release is a good point in history to do so.

If you do this, then you should make a public announcement indicating that *next* was rewound and rebuilt.

The same rewind and rebuild process may be followed for *seen*. A public announcement is not necessary since *seen* is a throw-away branch, as described above.

DISTRIBUTED WORKFLOWS

After the last section, you should know how to manage topics. In general, you will not be the only person working on the project, so you will have to share your work.

Roughly speaking, there are two important workflows: merge and patch. The important difference is that the merge workflow can propagate full history, including merges, while patches cannot. Both workflows can be used in parallel: in *git.git*, only subsystem maintainers use the merge workflow, while everyone else sends patches.

Note that the maintainer(s) may impose restrictions, such as "Signed-off-by" requirements, that all commits/patches submitted for inclusion must adhere to. Consult your project's documentation for more information.

1. Merge workflow

The merge workflow works by copying branches between upstream and downstream. Upstream can merge contributions into the official history; downstream base their work on the official history.

There are three main tools that can be used for this:

- Section G.3.103, “git-push(1)” copies your branches to a remote repository, usually to one that can be read by all involved parties;
- Section G.3.49, “git-fetch(1)” that copies remote branches to your repository; and
- Section G.3.102, “git-pull(1)” that does fetch and merge in one go.

Note the last point. Do *not* use *git pull* unless you actually want to merge the remote branch.

Getting changes out is easy:

git push <*remote*> <*branch*> and tell everyone where they can fetch from.

Example G.10. Push/pull: Publishing branches/topics

You will still have to tell people by other means, such as mail. (Git provides the Section G.3.116, “git-request-pull(1)” to send preformatted pull requests to upstream maintainers to simplify this task.)

If you just want to get the newest copies of the integration branches, staying up to date is easy too:

Use *git fetch <remote>* or *git remote update* to stay up to date.

Example G.11. Push/pull: Staying up to date

Then simply fork your topic branches from the stable remotes as explained earlier.

If you are a maintainer and would like to merge other people's topic branches to the integration branches, they will typically send a request to do so by mail. Such a request looks like

```
Please pull from
    <URL> <branch>
```

In that case, *git pull* can do the fetch and merge in one go, as follows.

git pull <URL> <branch>

Example G.12. Push/pull: Merging remote topics

Occasionally, the maintainer may get merge conflicts when they try to pull changes from downstream. In this case, they can ask downstream to do the merge and resolve the conflicts themselves (perhaps they will know better how to resolve them). It is one of the rare cases where downstream *should* merge from upstream.

2. Patch workflow

If you are a contributor that sends changes upstream in the form of emails, you should use topic branches as usual (see above). Then use Section G.3.54, “git-format-patch(1)” to generate the corresponding emails (highly recommended over manually formatting them because it makes the maintainer's life easier).

- *git format-patch -M upstream..topic* to turn them into preformatted patch files
- *git send-email --to=<recipient> <patches>*

Example G.13. format-patch/am: Publishing branches/topics

See the Section G.3.54, “git-format-patch(1)” and Section G.3.124, “git-send-email(1)” manpages for further usage notes.

If the maintainer tells you that your patch no longer applies to the current upstream, you will have to rebase your topic (you cannot use a merge because you cannot format-patch merges):

git pull --rebase <URL> <branch>

Example G.14. format-patch/am: Keeping topics up to date

You can then fix the conflicts during the rebase. Presumably you have not published your topic other than by mail, so rebasing it is not a problem.

If you receive such a patch series (as maintainer, or perhaps as a reader of the mailing list it was sent to), save the mails to files, create a new topic branch and use *git am* to import the commits:

git am < patch

Example G.15. format-patch/am: Importing patches

One feature worth pointing out is the three-way merge, which can help if you get conflicts: *git am -3* will use index information contained in patches to figure out the merge base. See Section G.3.3, “git-am(1)” for other options.

SEE ALSO

Section G.2.1, “gittutorial(7)”, Section G.3.103, “git-push(1)”, Section G.3.102, “git-pull(1)”, Section G.3.86, “git-merge(1)”, Section G.3.107, “git-rebase(1)”, Section G.3.54, “git-format-patch(1)”, Section G.3.124, “git-send-email(1)”, Section G.3.3, “git-am(1)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.4.19. gitglossary(7)

2024-07-02

Revision History

2024-07-02

NAME

gitglossary - A Git Glossary

SYNOPSIS

*

DESCRIPTION

alternate object database

Via the alternates mechanism, a repository can inherit part of its object database from another object database, which is called an "alternate".

bare repository

A bare repository is normally an appropriately named directory with a *.git* suffix that does not have a locally checked-out copy of any of the files under revision control. That is, all of the Git administrative and control files that would normally be present in the hidden *.git* sub-directory are directly present in the *repository.git* directory instead, and no other files are present and checked out. Usually publishers of public repositories make bare repositories available.

blob object

Untyped object, e.g. the contents of a file.

branch

A "branch" is a line of development. The most recent commit on a branch is referred to as the tip of that branch. The tip of the branch is referenced by a branch head, which moves forward as additional development is done on the branch. A single Git repository can track an arbitrary number of branches, but your working tree is associated with just one of them (the "current" or "checked out" branch), and HEAD points to that branch.

cache

Obsolete for: index.

chain

A list of objects, where each object in the list contains a reference to its successor (for example, the successor of a commit could be one of its parents).

changeset

BitKeeper/cvsps speak for "commit". Since Git does not store changes, but states, it really does not make sense to use the term "changesets" with Git.

checkout

The action of updating all or part of the working tree with a tree object or blob from the object database, and updating the index and HEAD if the whole working tree has been pointed at a new branch.

cherry-picking

In SCM jargon, "cherry pick" means to choose a subset of changes out of a series of changes (typically commits) and record them as a new series of changes on top of a different codebase. In Git, this is performed by the "git cherry-pick" command to extract the change introduced by an existing commit and to record it based on the tip of the current branch as a new commit.

clean

A working tree is clean, if it corresponds to the revision referenced by the current head. Also see "dirty".

commit

As a noun: A single point in the Git history; the entire history of a project is represented as a set of interrelated commits. The word "commit" is often used by Git in the same places other revision control systems use the words "revision" or "version". Also used as a short hand for commit object.

As a verb: The action of storing a new snapshot of the project's state in the Git history, by creating a new commit representing the current state of the index and advancing HEAD to point at the new commit.

commit graph concept, representations and usage

A synonym for the DAG structure formed by the commits in the object database, referenced by branch tips, using their chain of linked commits. This structure is the definitive commit graph. The graph can be represented in other ways, e.g. the "commit-graph" file.

commit-graph file

The "commit-graph" (normally hyphenated) file is a supplemental representation of the commit graph which accelerates commit graph walks. The "commit-graph" file is stored either in the .git/objects/info directory or in the info directory of an alternate object database.

commit object

An object which contains the information about a particular revision, such as parents, committer, author, date and the tree object which corresponds to the top directory of the stored revision.

commit-ish (also committish)

A commit object or an object that can be recursively dereferenced to a commit object. The following are all commit-ishes: a commit object, a tag object that points to a commit object, a tag object that points to a tag object that points to a commit object, etc.

core Git

Fundamental data structures and utilities of Git. Exposes only limited source code management tools.

DAG

Directed acyclic graph. The commit objects form a directed acyclic graph, because they have parents (directed), and the graph of commit objects is acyclic (there is no chain which begins and ends with the same object).

dangling object

An unreachable object which is not reachable even from other unreachable objects; a dangling object has no references to it from any reference or object in the repository.

dereference

Referring to a symbolic ref: the action of accessing the reference pointed at by a symbolic ref. Recursive dereferencing involves repeating the aforementioned process on the resulting ref until a non-symbolic reference is found.

Referring to a tag object: the action of accessing the object a tag points at. Tags are recursively dereferenced by repeating the operation on the result object until the result has either a specified object type (where applicable) or any non-"tag" object type. A synonym for "recursive dereference" in the context of tags is "peel".

Referring to a commit object: the action of accessing the commit's tree object. Commits cannot be dereferenced recursively.

Unless otherwise specified, "dereferencing" as it used in the context of Git commands or protocols is implicitly recursive.

detached HEAD

Normally the HEAD stores the name of a branch, and commands that operate on the history HEAD represents operate on the history leading to the tip of the branch the HEAD points at. However, Git also allows you to check out an arbitrary commit that isn't necessarily the tip of any particular branch. The HEAD in such a state is called "detached".

Note that commands that operate on the history of the current branch (e.g. *git commit* to build a new history on top of it) still work while the HEAD is detached. They update the HEAD to point at the tip of the updated history without affecting any branch. Commands that update or inquire information *about* the current branch (e.g. *git branch --set-upstream-to* that sets what remote-tracking branch the current branch integrates with) obviously do not work, as there is no (real) current branch to ask about in this state.

directory

The list you get with "ls" :-)

dirty

A working tree is said to be "dirty" if it contains modifications which have not been committed to the current branch.

evil merge

An evil merge is a merge that introduces changes that do not appear in any parent.

fast-forward

A fast-forward is a special type of merge where you have a revision and you are "merging" another branch's changes that happen to be a descendant of what you have. In such a case, you do not make a new merge commit but instead just update your branch to point at the same revision as the branch you are merging. This will happen frequently on a remote-tracking branch of a remote repository.

fetch

Fetching a branch means to get the branch's head ref from a remote repository, to find out which objects are missing from the local object database, and to get them, too. See also Section G.3.49, “git-fetch(1)”.

file system

Linus Torvalds originally designed Git to be a user space file system, i.e. the infrastructure to hold files and directories. That ensured the efficiency and speed of Git.

Git archive

Synonym for repository (for arch people).

gitfile

A plain file *.git* at the root of a working tree that points at the directory that is the real repository. For proper use see Section G.3.159, “git-worktree(1)” or Section G.3.141, “git-submodule(1)”. For syntax see Section G.4.13, “gitrepository-layout(5)”.

grafts

Grafts enable two otherwise different lines of development to be joined together by recording fake ancestry information for commits. This way you can make Git pretend the set of parents a commit has is different from what was recorded when the commit was created. Configured via the *.git/info/grafts* file.

Note that the grafts mechanism is outdated and can lead to problems transferring objects between repositories; see Section G.3.114, “git-replace(1)” for a more flexible and robust system to do the same thing.

hash

In Git's context, synonym for object name.

head

A named reference to the commit at the tip of a branch. Heads are stored in a file in *$GIT_DIR/refs/heads/* directory, except when using packed refs. (See Section G.3.98, “git-pack-refs(1)”.)

HEAD

The current branch. In more detail: Your working tree is normally derived from the state of the tree referred to by HEAD. HEAD is a reference to one of the heads in your repository, except when using a detached HEAD, in which case it directly references an arbitrary commit.

head ref

A synonym for head.

hook

During the normal execution of several Git commands, call-outs are made to optional scripts that allow a developer to add functionality or checking. Typically, the hooks allow for a command to be pre-verified and potentially aborted, and allow for a post-notification after the operation is done. The hook scripts are found in the *$GIT_DIR/hooks/* directory, and are enabled by simply removing the *.sample* suffix from the filename. In earlier versions of Git you had to make them executable.

index

A collection of files with stat information, whose contents are stored as objects. The index is a stored version of your working tree. Truth be told, it can also contain a second, and even a third version of a working tree, which are used when merging.

index entry

The information regarding a particular file, stored in the index. An index entry can be unmerged, if a merge was started, but not yet finished (i.e. if the index contains multiple versions of that file).

master

The default development branch. Whenever you create a Git repository, a branch named "master" is created, and becomes the active branch. In most cases, this contains the local development, though that is purely by convention and is not required.

merge

As a verb: To bring the contents of another branch (possibly from an external repository) into the current branch. In the case where the merged-in branch is from a different repository, this is done by first fetching the remote branch and then merging the result into the current branch. This combination of fetch and merge operations is called a pull. Merging is performed by an automatic process that identifies changes made since the branches diverged, and then applies all those changes together. In cases where changes conflict, manual intervention may be required to complete the merge.

As a noun: unless it is a fast-forward, a successful merge results in the creation of a new commit representing the result of the merge, and having as parents the tips of the merged branches. This commit is referred to as a "merge commit", or sometimes just a "merge".

object

The unit of storage in Git. It is uniquely identified by the SHA-1 of its contents. Consequently, an object cannot be changed.

object database

Stores a set of "objects", and an individual object is identified by its object name. The objects usually live in *$GIT_DIR/objects/*.

object identifier (oid)

Synonym for object name.

object name

The unique identifier of an object. The object name is usually represented by a 40 character hexadecimal string. Also colloquially called SHA-1.

object type

One of the identifiers "commit", "tree", "tag" or "blob" describing the type of an object.

octopus

To merge more than two branches.

orphan

The act of getting on a branch that does not exist yet (i.e., an unborn branch). After such an operation, the commit first created becomes a commit without a parent, starting a new history.

origin

The default upstream repository. Most projects have at least one upstream project which they track. By default *origin* is used for that purpose. New upstream updates will be fetched into remote-tracking branches named origin/name-of-upstream-branch, which you can see using *git branch -r*.

overlay

Only update and add files to the working directory, but don't delete them, similar to how *cp -R* would update the contents in the destination directory. This is the default mode in a checkout when checking out files from the index or a tree-ish. In contrast, no-overlay mode also deletes tracked files not present in the source, similar to *rsync --delete*.

pack

A set of objects which have been compressed into one file (to save space or to transmit them efficiently).

pack index

The list of identifiers, and other information, of the objects in a pack, to assist in efficiently accessing the contents of a pack.

pathspec

Pattern used to limit paths in Git commands.

Pathspecs are used on the command line of "git ls-files", "git ls-tree", "git add", "git grep", "git diff", "git checkout", and many other commands to limit the scope of operations to some subset of the tree or working tree. See the documentation of each command for whether paths are relative to the current directory or toplevel. The pathspec syntax is as follows:

- any path matches itself
- the pathspec up to the last slash represents a directory prefix. The scope of that pathspec is limited to that subtree.
- the rest of the pathspec is a pattern for the remainder of the pathname. Paths relative to the directory prefix will be matched against that pattern using fnmatch(3); in particular, * and *? can* match directory separators.

For example, Documentation/*.jpg will match all .jpg files in the Documentation subtree, including Documentation/chapter_1/figure_1.jpg.

A pathspec that begins with a colon *:* has special meaning. In the short form, the leading colon *:* is followed by zero or more "magic signature" letters (which optionally is terminated by another colon *:*), and the remainder is the pattern to match against the path. The "magic signature" consists of ASCII symbols that are neither alphanumeric, glob, regex special characters nor colon. The optional colon that terminates the "magic signature" can be omitted if the pattern begins with a character that does not belong to "magic signature" symbol set and is not a colon.

In the long form, the leading colon *:* is followed by an open parenthesis *(*, a comma-separated list of zero or more "magic words", and a close parentheses *)*, and the remainder is the pattern to match against the path.

A pathspec with only a colon means "there is no pathspec". This form should not be combined with other pathspec.

top

The magic word *top* (magic signature: */*) makes the pattern match from the root of the working tree, even when you are running the command from inside a subdirectory.

literal

Wildcards in the pattern such as * or *?* are treated as literal characters.

icase

Case insensitive match.

glob

Git treats the pattern as a shell glob suitable for consumption by fnmatch(3) with the FNM_PATHNAME flag: wildcards in the pattern will not match a / in the pathname. For example, "Documentation/*.html" matches "Documentation/git.html" but not "Documentation/ppc/ppc.html" or "tools/perf/Documentation/perf.html".

Two consecutive asterisks ("**") in patterns matched against full pathname may have special meaning:

- A leading "**" followed by a slash means match in all directories. For example, "***/foo*" matches file or directory "*foo*" anywhere, the same as pattern "*foo*". "***/foo/bar*" matches file or directory "*bar*" anywhere that is directly under directory "*foo*".

- A trailing "/**" matches everything inside. For example, "*abc/***" matches all files inside directory "abc", relative to the location of the *.gitignore* file, with infinite depth.
- A slash followed by two consecutive asterisks then a slash matches zero or more directories. For example, "*a/**/b*" matches "*a/b*", "*a/x/b*", "*a/x/y/b*" and so on.
- Other consecutive asterisks are considered invalid.

  Glob magic is incompatible with literal magic.

attr

After *attr:* comes a space separated list of "attribute requirements", all of which must be met in order for the path to be considered a match; this is in addition to the usual non-magic pathspec pattern matching. See Section G.4.2, “gitattributes(5)”.

Each of the attribute requirements for the path takes one of these forms:

- "*ATTR*" requires that the attribute *ATTR* be set.
- "*-ATTR*" requires that the attribute *ATTR* be unset.
- "*ATTR=VALUE*" requires that the attribute *ATTR* be set to the string *VALUE*.
- "*!ATTR*" requires that the attribute *ATTR* be unspecified.

  Note that when matching against a tree object, attributes are still obtained from working tree, not from the given tree object.

exclude

After a path matches any non-exclude pathspec, it will be run through all exclude pathspecs (magic signature: *!* or its synonym ^). If it matches, the path is ignored. When there is no non-exclude pathspec, the exclusion is applied to the result set as if invoked without any pathspec.

parent

A commit object contains a (possibly empty) list of the logical predecessor(s) in the line of development, i.e. its parents.

peel

The action of recursively dereferencing a tag object.

pickaxe

The term pickaxe refers to an option to the diffcore routines that help select changes that add or delete a given text string. With the *--pickaxe-all* option, it can be used to view the full changeset that introduced or removed, say, a particular line of text. See Section G.3.44, “git-diff(1)”.

plumbing

Cute name for core Git.

porcelain

Cute name for programs and program suites depending on core Git, presenting a high level access to core Git. Porcelains expose more of a SCM interface than the plumbing.

per-worktree ref

Refs that are per-worktree, rather than global. This is presently only HEAD and any refs that start with *refs/bisect/*, but might later include other unusual refs.

pseudoref

Pseudorefs are a class of files under *$GIT_DIR* which behave like refs for the purposes of rev-parse, but which are treated specially by git. Pseudorefs both have names that are all-caps, and always start with a line consisting of a SHA-1 followed by whitespace. So, HEAD is not a pseudoref, because it is sometimes a symbolic ref. They might optionally contain some additional data. *MERGE_HEAD* and *CHERRY_PICK_HEAD* are examples. Unlike per-worktree refs, these files cannot be symbolic refs, and never have reflogs. They also cannot be updated through the normal ref update machinery. Instead, they are updated by directly writing to the files. However, they can be read as if they were refs, so *git rev-parse MERGE_HEAD* will work.

pull

Pulling a branch means to fetch it and merge it. See also Section G.3.102, “git-pull(1)”.

push

Pushing a branch means to get the branch's head ref from a remote repository, find out if it is an ancestor to the branch's local head ref, and in that case, putting all objects, which are reachable from the local head ref, and which are missing from the remote repository, into the remote object database, and updating the remote head ref. If the remote head is not an ancestor to the local head, the push fails.

reachable

All of the ancestors of a given commit are said to be "reachable" from that commit. More generally, one object is reachable from another if we can reach the one from the other by a chain that follows tags to whatever they tag, commits to their parents or trees, and trees to the trees or blobs that they contain.

reachability bitmaps

Reachability bitmaps store information about the reachability of a selected set of commits in a packfile, or a multi-pack index (MIDX), to speed up object search. The bitmaps are stored in a ".bitmap" file. A repository may have at most one bitmap file in use. The bitmap file may belong to either one pack, or the repository's multi-pack index (if it exists).

rebase

To reapply a series of changes from a branch to a different base, and reset the head of that branch to the result.

ref

A name that begins with *refs/* (e.g. *refs/heads/master*) that points to an object name or another ref (the latter is called a symbolic ref). For convenience, a ref can sometimes be abbreviated when used as an argument to a Git command; see Section G.4.14, “gitrevisions(7)” for details. Refs are stored in the repository.

The ref namespace is hierarchical. Different subhierarchies are used for different purposes (e.g. the *refs/heads/* hierarchy is used to represent local branches).

There are a few special-purpose refs that do not begin with *refs/*. The most notable example is *HEAD*.

reflog

A reflog shows the local "history" of a ref. In other words, it can tell you what the 3rd last revision in *this* repository was, and what was the current state in *this* repository, yesterday 9:14pm. See Section G.3.109, “git-reflog(1)” for details.

refspec

A "refspec" is used by fetch and push to describe the mapping between remote ref and local ref.

remote repository

A repository which is used to track the same project but resides somewhere else. To communicate with remotes, see fetch or push.

remote-tracking branch

A ref that is used to follow changes from another repository. It typically looks like *refs/remotes/foo/bar* (indicating that it tracks a branch named *bar* in a remote named *foo*), and matches the right-hand-side of a configured fetch refspec. A remote-tracking branch should not contain direct modifications or have local commits made to it.

repository

A collection of refs together with an object database containing all objects which are reachable from the refs, possibly accompanied by meta data from one or more porcelains. A repository can share an object database with other repositories via alternates mechanism.

resolve

The action of fixing up manually what a failed automatic merge left behind.

revision

Synonym for commit (the noun).

rewind

To throw away part of the development, i.e. to assign the head to an earlier revision.

SCM

Source code management (tool).

SHA-1

"Secure Hash Algorithm 1"; a cryptographic hash function. In the context of Git used as a synonym for object name.

shallow clone

Mostly a synonym to shallow repository but the phrase makes it more explicit that it was created by running *git clone --depth=...* command.

shallow repository

A shallow repository has an incomplete history some of whose commits have parents cauterized away (in other words, Git is told to pretend that these commits do not have the parents, even though they are recorded in the commit object). This is sometimes useful when you are interested only in the recent history of a project even though the real history recorded in the upstream is much larger. A shallow repository is created by giving the *--depth* option to Section G.3.24, “git-clone(1)”, and its history can be later deepened with Section G.3.49, “git-fetch(1)”.

stash entry

An object used to temporarily store the contents of a dirty working directory and the index for future reuse.

special ref

A ref that has different semantics than normal refs. These refs can be accessed via normal Git commands but may not behave the same as a normal ref in some cases.

The following special refs are known to Git:

- "*FETCH_HEAD*" is written by Section G.3.49, “git-fetch(1)” or Section G.3.102, “git-pull(1)”. It may refer to multiple object IDs. Each object ID is annotated with metadata indicating where it was fetched from and its fetch status.

- "*MERGE_HEAD*" is written by Section G.3.86, “git-merge(1)” when resolving merge conflicts. It contains all commit IDs which are being merged.

submodule

A repository that holds the history of a separate project inside another repository (the latter of which is called superproject).

superproject

A repository that references repositories of other projects in its working tree as submodules. The superproject knows about the names of (but does not hold copies of) commit objects of the contained submodules.

symref

Symbolic reference: instead of containing the SHA-1 id itself, it is of the format *ref: refs/some/thing* and when referenced, it recursively dereferences to this reference. *HEAD* is a prime example of a symref. Symbolic references are manipulated with the Section G.3.143, “git-symbolic-ref(1)” command.

tag

A ref under *refs/tags/* namespace that points to an object of an arbitrary type (typically a tag points to either a tag or a commit object). In contrast to a head, a tag is not updated by the *commit* command. A Git tag has nothing to do with a Lisp tag (which would be called an object type in Git's context). A tag is most typically used to mark a particular point in the commit ancestry chain.

tag object

An object containing a ref pointing to another object, which can contain a message just like a commit object. It can also contain a (PGP) signature, in which case it is called a "signed tag object".

topic branch

A regular Git branch that is used by a developer to identify a conceptual line of development. Since branches are very easy and inexpensive, it is often desirable to have several small branches that each contain very well defined concepts or small incremental yet related changes.

tree

Either a working tree, or a tree object together with the dependent blob and tree objects (i.e. a stored representation of a working tree).

tree object

An object containing a list of file names and modes along with refs to the associated blob and/or tree objects. A tree is equivalent to a directory.

tree-ish (also treeish)

A tree object or an object that can be recursively dereferenced to a tree object. Dereferencing a commit object yields the tree object corresponding to the revision's top directory. The following are all tree-ishes: a commit-ish, a tree object, a tag object that points to a tree object, a tag object that points to a tag object that points to a tree object, etc.

unborn

The HEAD can point at a branch that does not yet exist and that does not have any commit on it yet, and such a branch is called an unborn branch. The most typical way users encounter an unborn branch is by creating a repository anew without cloning from elsewhere. The HEAD would point at the *main* (or *master*, depending on your configuration) branch that is yet to be born. Also some operations can get you on an unborn branch with their orphan option.

unmerged index

An index which contains unmerged index entries.

unreachable object

An object which is not reachable from a branch, tag, or any other reference.

upstream branch

The default branch that is merged into the branch in question (or the branch in question is rebased onto). It is configured via branch.<name>.remote and branch.<name>.merge. If the upstream branch of *A* is *origin/B* sometimes we say "*A* is tracking *origin/B*".

working tree

The tree of actual checked out files. The working tree normally contains the contents of the HEAD commit's tree, plus any local changes that you have made but not yet committed.

worktree

A repository can have zero (i.e. bare repository) or one or more worktrees attached to it. One "worktree" consists of a "working tree" and repository metadata, most of which are shared among other worktrees of a single repository, and some of which are maintained separately per worktree (e.g. the index, HEAD and pseudorefs like MERGE_HEAD, per-worktree refs and per-worktree configuration file).

SEE ALSO

Section G.2.1, “gittutorial(7)”, Section G.2.2, “gittutorial-2(7)”, Section G.2.4, “gitcvs-migration(7)”, Section G.2.5, “giteveryday(7)”, *The Git User's Manual* [https://www.kernel.org/pub/software/scm/git/docs/user-manual.html]

GIT

Part of the Section G.3.1, “git(1)” suite

G.5. File formats, protocols and other developer interfaces

G.5.1. gitformat-bundle(5)

2024-07-02
Revision History
2024-07-02

NAME

gitformat-bundle - The bundle file format

SYNOPSIS

*.bundle
*.bdl

DESCRIPTION

The Git bundle format is a format that represents both refs and Git objects. A bundle is a header in a format similar to Section G.3.133, “git-show-ref(1)” followed by a pack in *.pack format.

The format is created and read by the Section G.3.12, “git-bundle(1)” command, and supported by e.g. Section G.3.49, “git-fetch(1)” and Section G.3.24, “git-clone(1)”.

FORMAT

We will use ABNF notation to define the Git bundle format. See Section G.5.8, “gitprotocol-common(5)” for the details.

A v2 bundle looks like this:

```
bundle    = signature *prerequisite *reference LF pack
signature = "# v2 git bundle" LF

prerequisite = "-" obj-id SP comment LF
comment      = *CHAR
reference    = obj-id SP refname LF

pack         = ... ; packfile
```

A v3 bundle looks like this:

```
bundle    = signature *capability *prerequisite *reference LF pack
signature = "# v3 git bundle" LF

capability   = "@" key ["=" value] LF
prerequisite = "-" obj-id SP comment LF
comment      = *CHAR
reference    = obj-id SP refname LF
key          = 1*(ALPHA / DIGIT / "-")
value        = *(%01-09 / %0b-FF)

pack         = ... ; packfile
```

SEMANTICS

A Git bundle consists of several parts.

- "Capabilities", which are only in the v3 format, indicate functionality that the bundle requires to be read properly.
- "Prerequisites" list the objects that are NOT included in the bundle and the reader of the bundle MUST already have, in order to use the data in the bundle. The objects stored in the bundle may refer to prerequisite objects and anything reachable from them (e.g. a tree object in the bundle can reference a blob that is reachable from a prerequisite) and/or expressed as a delta against prerequisite objects.
- "References" record the tips of the history graph, iow, what the reader of the bundle CAN "git fetch" from it.
- "Pack" is the pack data stream "git fetch" would send, if you fetch from a repository that has the references recorded in the "References" above into a repository that has references pointing at the objects listed in "Prerequisites" above.

In the bundle format, there can be a comment following a prerequisite obj-id. This is a comment and it has no specific meaning. The writer of the bundle MAY put any string here. The reader of the bundle MUST ignore the comment.

1. Note on shallow clones and Git bundles

Note that the prerequisites do not represent a shallow-clone boundary. The semantics of the prerequisites and the shallow-clone boundaries are different, and the Git bundle v2 format cannot represent a shallow clone repository.

CAPABILITIES

Because there is no opportunity for negotiation, unknown capabilities cause *git bundle* to abort.

- *object-format* specifies the hash algorithm in use, and can take the same values as the *extensions.objectFormat* configuration value.
- *filter* specifies an object filter as in the *--filter* option in Section G.3.120, “git-rev-list(1)”. The resulting pack-file must be marked as a *.promisor* pack-file after it is unbundled.

GIT

Part of the Section G.3.1, “git(1)” suite

G.5.2. gitformat-chunk(5)

2024-07-02
Revision History
2024-07-02

NAME

gitformat-chunk - Chunk-based file formats

SYNOPSIS

Used by Section G.5.3, “gitformat-commit-graph(5)” and the "MIDX" format (see the pack format documentation in Section G.5.5, “gitformat-pack(5)”).

DESCRIPTION

Some file formats in Git use a common concept of "chunks" to describe sections of the file. This allows structured access to a large file by scanning a small "table of contents" for the remaining data. This common format is used by the *commit-graph* and *multi-pack-index* files. See the *multi-pack-index* format in Section G.5.5, “gitformat-pack(5)” and the *commit-graph* format in Section G.5.3, “gitformat-commit-graph(5)” for how they use the chunks to describe structured data.

A chunk-based file format begins with some header information custom to that format. That header should include enough information to identify the file type, format version, and number of chunks in the file. From this information, that file can determine the start of the chunk-based region.

The chunk-based region starts with a table of contents describing where each chunk starts and ends. This consists of (C+1) rows of 12 bytes each, where C is the number of chunks. Consider the following table:

```
| Chunk ID (4 bytes) | Chunk Offset (8 bytes) |
|--------------------|------------------------|
| ID[0]              | OFFSET[0]              |
| ...                | ...                    |
| ID[C]              | OFFSET[C]              |
| 0x0000             | OFFSET[C+1]            |
```

Each row consists of a 4-byte chunk identifier (ID) and an 8-byte offset. Each integer is stored in network-byte order.

The chunk identifier *ID[i]* is a label for the data stored within this file from *OFFSET[i]* (inclusive) to *OFFSET[i+1]* (exclusive). Thus, the size of the *i`th chunk is equal to the difference between `OFFSET[i+1]* and *OFFSET[i]*. This requires that the chunk data appears contiguously in the same order as the table of contents.

The final entry in the table of contents must be four zero bytes. This confirms that the table of contents is ending and provides the offset for the end of the chunk-based data.

Note: The chunk-based format expects that the file contains *at least* a trailing hash after *OFFSET[C+1]*.

Functions for working with chunk-based file formats are declared in *chunk-format.h*. Using these methods provide extra checks that assist developers when creating new file formats.

Writing chunk-based file formats

To write a chunk-based file format, create a *struct chunkfile* by calling *init_chunkfile()* and pass a *struct hashfile* pointer. The caller is responsible for opening the *hashfile* and writing header information so the file format is identifiable before the chunk-based format begins.

Then, call *add_chunk()* for each chunk that is intended for writing. This populates the *chunkfile* with information about the order and size of each chunk to write. Provide a *chunk_write_fn* function pointer to perform the write of the chunk data upon request.

Call *write_chunkfile()* to write the table of contents to the *hashfile* followed by each of the chunks. This will verify that each chunk wrote the expected amount of data so the table of contents is correct.

Finally, call *free_chunkfile()* to clear the *struct chunkfile* data. The caller is responsible for finalizing the *hashfile* by writing the trailing hash and closing the file.

Reading chunk-based file formats

To read a chunk-based file format, the file must be opened as a memory-mapped region. The chunk-format API expects that the entire file is mapped as a contiguous memory region.

Initialize a *struct chunkfile* pointer with *init_chunkfile(NULL)*.

After reading the header information from the beginning of the file, including the chunk count, call *read_table_of_contents()* to populate the *struct chunkfile* with the list of chunks, their offsets, and their sizes.

Extract the data information for each chunk using *pair_chunk()* or *read_chunk()*:

- *pair_chunk()* assigns a given pointer with the location inside the memory-mapped file corresponding to that chunk's offset. If the chunk does not exist, then the pointer is not modified.
- *read_chunk()* takes a *chunk_read_fn* function pointer and calls it with the appropriate initial pointer and size information. The function is not called if the chunk does not exist. Use this method to read chunks if you need to perform immediate parsing or if you need to execute logic based on the size of the chunk.

After calling these methods, call *free_chunkfile()* to clear the *struct chunkfile* data. This will not close the memory-mapped region. Callers are expected to own that data for the timeframe the pointers into the region are needed.

Examples

These file formats use the chunk-format API, and can be used as examples for future formats:

- **commit-graph:** see *write_commit_graph_file()* and *parse_commit_graph()* in *commit-graph.c* for how the chunk-format API is used to write and parse the commit-graph file format documented in the commit-graph file format in Section G.5.3, “gitformat-commit-graph(5)”.
- **multi-pack-index:** see *write_midx_internal()* and *load_multi_pack_index()* in *midx.c* for how the chunk-format API is used to write and parse the multi-pack-index file format documented in the multi-pack-index file format section of Section G.5.5, “gitformat-pack(5)”.

GIT

Part of the Section G.3.1, “git(1)” suite

G.5.3. gitformat-commit-graph(5)

2024-07-02
Revision History

2024-07-02

NAME

gitformat-commit-graph - Git commit-graph format

SYNOPSIS

$GIT_DIR/objects/info/commit-graph
$GIT_DIR/objects/info/commit-graphs/*

DESCRIPTION

The Git commit-graph stores a list of commit OIDs and some associated metadata, including:

- The generation number of the commit.
- The root tree OID.
- The commit date.
- The parents of the commit, stored using positional references within the graph file.
- The Bloom filter of the commit carrying the paths that were changed between the commit and its first parent, if requested.

These positional references are stored as unsigned 32-bit integers corresponding to the array position within the list of commit OIDs. Due to some special constants we use to track parents, we can store at most (1 << 30) + (1 << 29) + (1 << 28) - 1 (around 1.8 billion) commits.

Commit-graph files have the following format:

In order to allow extensions that add extra data to the graph, we organize the body into "chunks" and provide a binary lookup table at the beginning of the body. The header includes certain values, such as number of chunks and hash type.

All multi-byte numbers are in network byte order.

1. HEADER:

```
4-byte signature:
    The signature is: {'C', 'G', 'P', 'H'}

1-byte version number:
    Currently, the only valid version is 1.

1-byte Hash Version
    We infer the hash length (H) from this value:
      1 => SHA-1
      2 => SHA-256
    If the hash type does not match the repository's hash algorithm, the
    commit-graph file should be ignored with a warning presented to the
    user.

1-byte number (C) of "chunks"

1-byte number (B) of base commit-graphs
    We infer the length (H*B) of the Base Graphs chunk
    from this value.
```

2. CHUNK LOOKUP:

```
(C + 1) * 12 bytes listing the table of contents for the chunks:
    First 4 bytes describe the chunk id. Value 0 is a terminating label.
    Other 8 bytes provide the byte-offset in current file for chunk to
    start. (Chunks are ordered contiguously in the file, so you can infer
```

```
    the length using the next chunk position if necessary.) Each chunk
    ID appears at most once.

The CHUNK LOOKUP matches the table of contents from
the chunk-based file format, see linkgit:gitformat-chunk[5]

The remaining data in the body is described one chunk at a time, and
these chunks may be given in any order. Chunks are required unless
otherwise specified.
```

3. CHUNK DATA:

3.1. OID Fanout (ID: {*O*, *I*, *D*, *F*}) (256 * 4 bytes)

```
The ith entry, F[i], stores the number of OIDs with first
byte at most i. Thus F[255] stores the total
number of commits (N).
```

3.2. OID Lookup (ID: {*O*, *I*, *D*, *L*}) (N * H bytes)

```
The OIDs for all commits in the graph, sorted in ascending order.
```

3.3. Commit Data (ID: {*C*, *D*, *A*, *T* }) (N * (H + 16) bytes)

- The first H bytes are for the OID of the root tree.
- The next 8 bytes are for the positions of the first two parents of the ith commit. Stores value 0x70000000 if no parent in that position. If there are more than two parents, the second value has its most-significant bit on and the other bits store an array position into the Extra Edge List chunk.
- The next 8 bytes store the topological level (generation number v1) of the commit and the commit time in seconds since EPOCH. The generation number uses the higher 30 bits of the first 4 bytes, while the commit time uses the 32 bits of the second 4 bytes, along with the lowest 2 bits of the lowest byte, storing the 33rd and 34th bit of the commit time.

3.4. Generation Data (ID: {*G*, *D*, *A*, *2* }) (N * 4 bytes) [Optional]

- This list of 4-byte values store corrected commit date offsets for the commits, arranged in the same order as commit data chunk.
- If the corrected commit date offset cannot be stored within 31 bits, the value has its most-significant bit on and the other bits store the position of corrected commit date into the Generation Data Overflow chunk.
- Generation Data chunk is present only when commit-graph file is written by compatible versions of Git and in case of split commit-graph chains, the topmost layer also has Generation Data chunk.

3.5. Generation Data Overflow (ID: {*G*, *D*, *O*, *2* }) [Optional]

- This list of 8-byte values stores the corrected commit date offsets for commits with corrected commit date offsets that cannot be stored within 31 bits.
- Generation Data Overflow chunk is present only when Generation Data chunk is present and atleast one corrected commit date offset cannot be stored within 31 bits.

3.6. Extra Edge List (ID: {*E*, *D*, *G*, *E*}) [Optional]

```
This list of 4-byte values store the second through nth parents for
all octopus merges. The second parent value in the commit data stores
an array position within this list along with the most-significant bit
```

```
on. Starting at that array position, iterate through this list of commit
positions for the parents until reaching a value with the most-significant
bit on. The other bits correspond to the position of the last parent.
```

3.7. Bloom Filter Index (ID: {*B*, *I*, *D*, *X*}) (N * 4 bytes) [Optional]

- The ith entry, BIDX[i], stores the number of bytes in all Bloom filters from commit 0 to commit i (inclusive) in lexicographic order. The Bloom filter for the i-th commit spans from BIDX[i-1] to BIDX[i] (plus header length), where BIDX[-1] is 0.
- The BIDX chunk is ignored if the BDAT chunk is not present.

3.8. Bloom Filter Data (ID: {*B*, *D*, *A*, *T*}) [Optional]

- It starts with header consisting of three unsigned 32-bit integers:
  - Version of the hash algorithm being used. We currently only support value 1 which corresponds to the 32-bit version of the murmur3 hash implemented exactly as described in https://en.wikipedia.org/wiki/MurmurHash#Algorithm and the double hashing technique using seed values 0x293ae76f and 0x7e646e2 as described in https://doi.org/10.1007/978-3-540-30494-4_26 "Bloom Filters in Probabilistic Verification"
  - The number of times a path is hashed and hence the number of bit positions that cumulatively determine whether a file is present in the commit.
  - The minimum number of bits *b* per entry in the Bloom filter. If the filter contains *n* entries, then the filter size is the minimum number of 64-bit words that contain n*b bits.
- The rest of the chunk is the concatenation of all the computed Bloom filters for the commits in lexicographic order.
- Note: Commits with no changes or more than 512 changes have Bloom filters of length one, with either all bits set to zero or one respectively.
- The BDAT chunk is present if and only if BIDX is present.

3.9. Base Graphs List (ID: {*B*, *A*, *S*, *E*}) [Optional]

```
This list of H-byte hashes describe a set of B commit-graph files that
form a commit-graph chain. The graph position for the ith commit in this
file's OID Lookup chunk is equal to i plus the number of commits in all
base graphs.  If B is non-zero, this chunk must exist.
```

4. TRAILER:

```
H-byte HASH-checksum of all of the above.
```

Historical Notes:

The Generation Data (GDA2) and Generation Data Overflow (GDO2) chunks have the number *2* in their chunk IDs because a previous version of Git wrote possibly erroneous data in these chunks with the IDs "GDAT" and "GDOV". By changing the IDs, newer versions of Git will silently ignore those older chunks and write the new information without trusting the incorrect data.

GIT

Part of the Section G.3.1, “git(1)” suite

G.5.4. gitformat-index(5)

2024-07-02
Revision History

2024-07-02

NAME

gitformat-index - Git index format

SYNOPSIS

$GIT_DIR/index

DESCRIPTION

Git index format

The Git index file has the following format

```
All binary numbers are in network byte order.
In a repository using the traditional SHA-1, checksums and object IDs
(object names) mentioned below are all computed using SHA-1.  Similarly,
in SHA-256 repositories, these values are computed using SHA-256.
Version 2 is described here unless stated otherwise.
```

- A 12-byte header consisting of

  ```
  4-byte signature:
    The signature is { 'D', 'I', 'R', 'C' } (stands for "dircache")

  4-byte version number:
    The current supported versions are 2, 3 and 4.

  32-bit number of index entries.
  ```

- A number of sorted index entries (see below).

- Extensions

  ```
  Extensions are identified by signature. Optional extensions can
  be ignored if Git does not understand them.

  4-byte extension signature. If the first byte is 'A'..'Z' the
  extension is optional and can be ignored.

  32-bit size of the extension

  Extension data
  ```

- Hash checksum over the content of the index file before this checksum.

Index entry

```
Index entries are sorted in ascending order on the name field,
interpreted as a string of unsigned bytes (i.e. memcmp() order, no
localization, no special casing of directory separator '/'). Entries
with the same name are sorted by their stage field.

An index entry typically represents a file. However, if sparse-checkout
is enabled in cone mode (`core.sparseCheckoutCone` is enabled) and the
`extensions.sparseIndex` extension is enabled, then the index may
contain entries for directories outside of the sparse-checkout definition.
These entries have mode `040000`, include the `SKIP_WORKTREE` bit, and
the path ends in a directory separator.
```

```
32-bit ctime seconds, the last time a file's metadata changed
  this is stat(2) data

32-bit ctime nanosecond fractions
  this is stat(2) data

32-bit mtime seconds, the last time a file's data changed
  this is stat(2) data

32-bit mtime nanosecond fractions
  this is stat(2) data

32-bit dev
  this is stat(2) data

32-bit ino
  this is stat(2) data

32-bit mode, split into (high to low bits)

16-bit unused, must be zero

4-bit object type
  valid values in binary are 1000 (regular file), 1010 (symbolic link)
  and 1110 (gitlink)

3-bit unused, must be zero

9-bit unix permission. Only 0755 and 0644 are valid for regular files.
Symbolic links and gitlinks have value 0 in this field.

32-bit uid
  this is stat(2) data

32-bit gid
  this is stat(2) data

32-bit file size
  This is the on-disk size from stat(2), truncated to 32-bit.

Object name for the represented object

A 16-bit 'flags' field split into (high to low bits)

1-bit assume-valid flag

1-bit extended flag (must be zero in version 2)

2-bit stage (during merge)

12-bit name length if the length is less than 0xFFF; otherwise 0xFFF
is stored in this field.

(Version 3 or later) A 16-bit field, only applicable if the
"extended flag" above is 1, split into (high to low bits).

1-bit reserved for future

1-bit skip-worktree flag (used by sparse checkout)

1-bit intent-to-add flag (used by "git add -N")
```

```
13-bit unused, must be zero

Entry path name (variable length) relative to top level directory
  (without leading slash). '/' is used as path separator. The special
  path components ".", ".." and ".git" (without quotes) are disallowed.
  Trailing slash is also disallowed.

The exact encoding is undefined, but the '.' and '/' characters
are encoded in 7-bit ASCII and the encoding cannot contain a NUL
byte (iow, this is a UNIX pathname).

(Version 4) In version 4, the entry path name is prefix-compressed
  relative to the path name for the previous entry (the very first
  entry is encoded as if the path name for the previous entry is an
  empty string).  At the beginning of an entry, an integer N in the
  variable width encoding (the same encoding as the offset is encoded
  for OFS_DELTA pack entries; see linkgit:gitformat-pack[5]) is stored,
 followed
  by a NUL-terminated string S.  Removing N bytes from the end of the
  path name for the previous entry, and replacing it with the string S
  yields the path name for this entry.

1-8 nul bytes as necessary to pad the entry to a multiple of eight bytes
while keeping the name NUL-terminated.

(Version 4) In version 4, the padding after the pathname does not
exist.

Interpretation of index entries in split index mode is completely
different. See below for details.
```

Extensions

1. Cache tree

```
Since the index does not record entries for directories, the cache
entries cannot describe tree objects that already exist in the object
database for regions of the index that are unchanged from an existing
commit. The cache tree extension stores a recursive tree structure that
describes the trees that already exist and completely match sections of
the cache entries. This speeds up tree object generation from the index
for a new commit by only computing the trees that are "new" to that
commit. It also assists when comparing the index to another tree, such
as `HEAD^{tree}`, since sections of the index can be skipped when a tree
comparison demonstrates equality.

The recursive tree structure uses nodes that store a number of cache
entries, a list of subnodes, and an object ID (OID). The OID references
the existing tree for that node, if it is known to exist. The subnodes
correspond to subdirectories that themselves have cache tree nodes. The
number of cache entries corresponds to the number of cache entries in
the index that describe paths within that tree's directory.

The extension tracks the full directory structure in the cache tree
extension, but this is generally smaller than the full cache entry list.

When a path is updated in index, Git invalidates all nodes of the
recursive cache tree corresponding to the parent directories of that
path. We store these tree nodes as being "invalid" by using "-1" as the
```

```
number of cache entries. Invalid nodes still store a span of index
entries, allowing Git to focus its efforts when reconstructing a full
cache tree.

The signature for this extension is { 'T', 'R', 'E', 'E' }.

A series of entries fill the entire extension; each of which
consists of:
```

- NUL-terminated path component (relative to its parent directory);
- ASCII decimal number of entries in the index that is covered by the tree this entry represents (entry_count);
- A space (ASCII 32);
- ASCII decimal number that represents the number of subtrees this tree has;
- A newline (ASCII 10); and
- Object name for the object that would result from writing this span of index as a tree.

```
An entry can be in an invalidated state and is represented by having
a negative number in the entry_count field. In this case, there is no
object name and the next entry starts immediately after the newline.
When writing an invalid entry, -1 should always be used as entry_count.

The entries are written out in the top-down, depth-first order.  The
first entry represents the root level of the repository, followed by the
first subtree--let's call this A--of the root level (with its name
relative to the root level), followed by the first subtree of A (with
its name relative to A), and so on. The specified number of subtrees
indicates when the current level of the recursive stack is complete.
```

2. Resolve undo

```
A conflict is represented in the index as a set of higher stage entries.
When a conflict is resolved (e.g. with "git add path"), these higher
stage entries will be removed and a stage-0 entry with proper resolution
is added.

When these higher stage entries are removed, they are saved in the
resolve undo extension, so that conflicts can be recreated (e.g. with
"git checkout -m"), in case users want to redo a conflict resolution
from scratch.

The signature for this extension is { 'R', 'E', 'U', 'C' }.

A series of entries fill the entire extension; each of which
consists of:
```

- NUL-terminated pathname the entry describes (relative to the root of the repository, i.e. full pathname);
- Three NUL-terminated ASCII octal numbers, entry mode of entries in stage 1 to 3 (a missing stage is represented by "0" in this field); and
- At most three object names of the entry in stages from 1 to 3 (nothing is written for a missing stage).

3. Split index

```
In split index mode, the majority of index entries could be stored
```

```
in a separate file. This extension records the changes to be made on
top of that to produce the final index.

The signature for this extension is { 'l', 'i', 'n', 'k' }.

The extension consists of:
```

- Hash of the shared index file. The shared index file path is $GIT_DIR/sharedindex.<hash>. If all bits are zero, the index does not require a shared index file.
- An ewah-encoded delete bitmap, each bit represents an entry in the shared index. If a bit is set, its corresponding entry in the shared index will be removed from the final index. Note, because a delete operation changes index entry positions, but we do need original positions in replace phase, it's best to just mark entries for removal, then do a mass deletion after replacement.
- An ewah-encoded replace bitmap, each bit represents an entry in the shared index. If a bit is set, its corresponding entry in the shared index will be replaced with an entry in this index file. All replaced entries are stored in sorted order in this index. The first "1" bit in the replace bitmap corresponds to the first index entry, the second "1" bit to the second entry and so on. Replaced entries may have empty path names to save space.

  ```
  The remaining index entries after replaced ones will be added to the
  final index. These added entries are also sorted by entry name then
  stage.
  ```

Untracked cache

```
Untracked cache saves the untracked file list and necessary data to
verify the cache. The signature for this extension is { 'U', 'N',
'T', 'R' }.

The extension starts with
```

- A sequence of NUL-terminated strings, preceded by the size of the sequence in variable width encoding. Each string describes the environment where the cache can be used.
- Stat data of $GIT_DIR/info/exclude. See "Index entry" section from ctime field until "file size".
- Stat data of core.excludesFile
- 32-bit dir_flags (see struct dir_struct)
- Hash of $GIT_DIR/info/exclude. A null hash means the file does not exist.
- Hash of core.excludesFile. A null hash means the file does not exist.
- NUL-terminated string of per-dir exclude file name. This usually is ".gitignore".
- The number of following directory blocks, variable width encoding. If this number is zero, the extension ends here with a following NUL.
- A number of directory blocks in depth-first-search order, each consists of
- The number of untracked entries, variable width encoding.
- The number of sub-directory blocks, variable width encoding.
- The directory name terminated by NUL.
- A number of untracked file/dir names terminated by NUL.

The remaining data of each directory block is grouped by type:

- An ewah bitmap, the n-th bit marks whether the n-th directory has valid untracked cache entries.
- An ewah bitmap, the n-th bit records "check-only" bit of read_directory_recursive() for the n-th directory.
- An ewah bitmap, the n-th bit indicates whether hash and stat data is valid for the n-th directory and exists in the next data.
- An array of stat data. The n-th data corresponds with the n-th "one" bit in the previous ewah bitmap.
- An array of hashes. The n-th hash corresponds with the n-th "one" bit in the previous ewah bitmap.
- One NUL.

File System Monitor cache

```
The file system monitor cache tracks files for which the core.fsmonitor
hook has told us about changes.  The signature for this extension is
{ 'F', 'S', 'M', 'N' }.

The extension starts with
```

- 32-bit version number: the current supported versions are 1 and 2.
- (Version 1) 64-bit time: the extension data reflects all changes through the given time which is stored as the nanoseconds elapsed since midnight, January 1, 1970.
- (Version 2) A null terminated string: an opaque token defined by the file system monitor application. The extension data reflects all changes relative to that token.
- 32-bit bitmap size: the size of the CE_FSMONITOR_VALID bitmap.
- An ewah bitmap, the n-th bit indicates whether the n-th index entry is not CE_FSMONITOR_VALID.

End of Index Entry

```
The End of Index Entry (EOIE) is used to locate the end of the variable
length index entries and the beginning of the extensions. Code can take
advantage of this to quickly locate the index extensions without having
to parse through all of the index entries.

Because it must be able to be loaded before the variable length cache
entries and other index extensions, this extension must be written last.
The signature for this extension is { 'E', 'O', 'I', 'E' }.

The extension consists of:
```

- 32-bit offset to the end of the index entries
- Hash over the extension types and their sizes (but not their contents). E.g. if we have "TREE" extension that is N-bytes long, "REUC" extension that is M-bytes long, followed by "EOIE", then the hash would be:

```
Hash("TREE" + <binary-representation-of-N> +
        "REUC" + <binary-representation-of-M>)
```

Index Entry Offset Table

```
The Index Entry Offset Table (IEOT) is used to help address the CPU
cost of loading the index by enabling multi-threading the process of
converting cache entries from the on-disk format to the in-memory format.
The signature for this extension is { 'I', 'E', 'O', 'T' }.

The extension consists of:
```

- 32-bit version (currently 1)
- A number of index offset entries each consisting of:
- 32-bit offset from the beginning of the file to the first cache entry in this block of entries.
- 32-bit count of cache entries in this block

Sparse Directory Entries

```
When using sparse-checkout in cone mode, some entire directories within
the index can be summarized by pointing to a tree object instead of the
entire expanded list of paths within that tree. An index containing such
entries is a "sparse index". Index format versions 4 and less were not
implemented with such entries in mind. Thus, for these versions, an
index containing sparse directory entries will include this extension
with signature { 's', 'd', 'i', 'r' }. Like the split-index extension,
tools should avoid interacting with a sparse index unless they understand
this extension.
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.5.5. gitformat-pack(5)

2024-07-02
Revision History

2024-07-02

NAME

gitformat-pack - Git pack format

SYNOPSIS

$GIT_DIR/objects/pack/pack-**.{pack,idx}**
$GIT_DIR/objects/pack/pack-.rev
$GIT_DIR/objects/pack/pack-*.mtimes
$GIT_DIR/objects/pack/multi-pack-index

DESCRIPTION

The Git pack format is how Git stores most of its primary repository data. Over the lifetime of a repository, loose objects (if any) and smaller packs are consolidated into larger pack(s). See Section G.3.58, “git-gc(1)” and Section G.3.96, “git-pack-objects(1)”.

The pack format is also used over-the-wire, see e.g. Section G.5.11, “gitprotocol-v2(5)”, as well as being a part of other container formats in the case of Section G.5.1, “gitformat-bundle(5)”.

Checksums and object IDs

In a repository using the traditional SHA-1, pack checksums, index checksums, and object IDs (object names) mentioned below are all computed using SHA-1. Similarly, in SHA-256 repositories, these values are computed using SHA-256.

pack-*.pack files have the following format:

- A header appears at the beginning and consists of the following:

```
4-byte signature:
    The signature is: {'P', 'A', 'C', 'K'}
```

```
4-byte version number (network byte order):
    Git currently accepts version number 2 or 3 but
    generates version 2 only.

4-byte number of objects contained in the pack (network byte order)

Observation: we cannot have more than 4G versions ;-) and
more than 4G objects in a pack.
```

- The header is followed by a number of object entries, each of which looks like this:

```
(undeltified representation)
n-byte type and length (3-bit type, (n-1)*7+4-bit length)
compressed data

(deltified representation)
n-byte type and length (3-bit type, (n-1)*7+4-bit length)
base object name if OBJ_REF_DELTA or a negative relative
    offset from the delta object's position in the pack if this
    is an OBJ_OFS_DELTA object
compressed delta data

Observation: the length of each object is encoded in a variable
length format and is not constrained to 32-bit or anything.
```

- The trailer records a pack checksum of all of the above.

1. Object types

Valid object types are:

- OBJ_COMMIT (1)
- OBJ_TREE (2)
- OBJ_BLOB (3)
- OBJ_TAG (4)
- OBJ_OFS_DELTA (6)
- OBJ_REF_DELTA (7)

Type 5 is reserved for future expansion. Type 0 is invalid.

2. Size encoding

This document uses the following "size encoding" of non-negative integers: From each byte, the seven least significant bits are used to form the resulting integer. As long as the most significant bit is 1, this process continues; the byte with MSB 0 provides the last seven bits. The seven-bit chunks are concatenated. Later values are more significant.

This size encoding should not be confused with the "offset encoding", which is also used in this document.

3. Deltified representation

Conceptually there are only four object types: commit, tree, tag and blob. However to save space, an object could be stored as a "delta" of another "base" object. These representations are assigned new types ofs-delta and ref-delta, which is only valid in a pack file.

Both ofs-delta and ref-delta store the "delta" to be applied to another object (called *base object*) to reconstruct the object. The difference between them is, ref-delta directly encodes base object name. If the base object is in the same pack, ofs-delta encodes the offset of the base object in the pack instead.

The base object could also be deltified if it's in the same pack. Ref-delta can also refer to an object outside the pack (i.e. the so-called "thin pack"). When stored on disk however, the pack should be self contained to avoid cyclic dependency.

The delta data starts with the size of the base object and the size of the object to be reconstructed. These sizes are encoded using the size encoding from above. The remainder of the delta data is a sequence of instructions to reconstruct the object from the base object. If the base object is deltified, it must be converted to canonical form first. Each instruction appends more and more data to the target object until it's complete. There are two supported instructions so far: one for copying a byte range from the source object and one for inserting new data embedded in the instruction itself.

Each instruction has variable length. Instruction type is determined by the seventh bit of the first octet. The following diagrams follow the convention in RFC 1951 (Deflate compressed data format).

3.1. Instruction to copy from base object

```
+----------+---------+---------+---------+---------+-------+-------+-------
+
| 1xxxxxxx | offset1 | offset2 | offset3 | offset4 | size1 | size2 | size3
 |
+----------+---------+---------+---------+---------+-------+-------+-------
+
```

This is the instruction format to copy a byte range from the source object. It encodes the offset to copy from and the number of bytes to copy. Offset and size are in little-endian order.

All offset and size bytes are optional. This is to reduce the instruction size when encoding small offsets or sizes. The first seven bits in the first octet determine which of the next seven octets is present. If bit zero is set, offset1 is present. If bit one is set offset2 is present and so on.

Note that a more compact instruction does not change offset and size encoding. For example, if only offset2 is omitted like below, offset3 still contains bits 16-23. It does not become offset2 and contains bits 8-15 even if it's right next to offset1.

```
+----------+---------+---------+
| 10000101 | offset1 | offset3 |
+----------+---------+---------+
```

In its most compact form, this instruction only takes up one byte (0x80) with both offset and size omitted, which will have default values zero. There is another exception: size zero is automatically converted to 0x10000.

3.2. Instruction to add new data

```
+----------+============+
| 0xxxxxxx |    data    |
+----------+============+
```

This is the instruction to construct the target object without the base object. The following data is appended to the target object. The first seven bits of the first octet determine the size of data in bytes. The size must be non-zero.

3.3. Reserved instruction

```
+----------+============
| 00000000 |
+----------+============
```

This is the instruction reserved for future expansion.

Original (version 1) pack-*.idx files have the following format:

- The header consists of 256 4-byte network byte order integers. N-th entry of this table records the number of objects in the corresponding pack, the first byte of whose object name is less than or equal to N. This is called the *first-level fan-out* table.
- The header is followed by sorted 24-byte entries, one entry per object in the pack. Each entry is:

  ```
  4-byte network byte order integer, recording where the
  object is stored in the packfile as the offset from the
  beginning.

  one object name of the appropriate size.
  ```

- The file is concluded with a trailer:

  ```
  A copy of the pack checksum at the end of the corresponding
  packfile.

  Index checksum of all of the above.
  ```

Pack Idx file:

```
        --  +--------------------------------+
fanout      | fanout[0] = 2 (for example)    |-.
table       +--------------------------------+ |
            | fanout[1]                      | |
            +--------------------------------+ |
            | fanout[2]                      | |
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ |
            | fanout[255] = total objects    |---.
        --  +--------------------------------+ | |
main        | offset                         | | |
index       | object name 00XXXXXXXXXXXXXXXX | | |
table       +--------------------------------+ | |
            | offset                         | | |
            | object name 00XXXXXXXXXXXXXXXX | | |
            +--------------------------------+<+ |
          .-| offset                         |   |
          | | object name 01XXXXXXXXXXXXXXXX |   |
          | +--------------------------------+   |
          | | offset                         |   |
          | | object name 01XXXXXXXXXXXXXXXX |   |
          | ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~   |
          | | offset                         |   |
          | | object name FFXXXXXXXXXXXXXXXX |   |
        --| +--------------------------------+<--+
trailer   | | packfile checksum              |
          | +--------------------------------+
          | | idxfile checksum               |
          | +--------------------------------+
          .-------.
                  |
Pack file entry: <+

packed object header:
   1-byte size extension bit (MSB)
          type (next 3 bit)
```

```
            size0 (lower 4-bit)
   n-byte sizeN (as long as MSB is set, each 7-bit)
           size0..sizeN form 4+7+7+..+7 bit integer, size0
           is the least significant part, and sizeN is the
           most significant part.
packed object data:
   If it is not DELTA, then deflated bytes (the size above
           is the size before compression).
   If it is REF_DELTA, then
     base object name (the size above is the
           size of the delta data that follows).
     delta data, deflated.
   If it is OFS_DELTA, then
     n-byte offset (see below) interpreted as a negative
           offset from the type-byte of the header of the
           ofs-delta entry (the size above is the size of
           the delta data that follows).
     delta data, deflated.

offset encoding:
     n bytes with MSB set in all but the last one.
     The offset is then the number constructed by
     concatenating the lower 7 bit of each byte, and
     for n >= 2 adding 2^7 + 2^14 + ... + 2^(7*(n-1))
     to the result.
```

Version 2 pack-*.idx files support packs larger than 4 GiB, and

```
have some other reorganizations.  They have the format:
```

- A 4-byte magic number *\377tOc* which is an unreasonable fanout[0] value.
- A 4-byte version number (= 2)
- A 256-entry fan-out table just like v1.
- A table of sorted object names. These are packed together without offset values to reduce the cache footprint of the binary search for a specific object name.
- A table of 4-byte CRC32 values of the packed object data. This is new in v2 so compressed data can be copied directly from pack to pack during repacking without undetected data corruption.
- A table of 4-byte offset values (in network byte order). These are usually 31-bit pack file offsets, but large offsets are encoded as an index into the next table with the msbit set.
- A table of 8-byte offset entries (empty for pack files less than 2 GiB). Pack files are organized with heavily used objects toward the front, so most object references should not need to refer to this table.
- The same trailer as a v1 pack file:

  ```
  A copy of the pack checksum at the end of the
  corresponding packfile.

  Index checksum of all of the above.
  ```

pack-*.rev files have the format:

- A 4-byte magic number *0x52494458* (*RIDX*).
- A 4-byte version identifier (= 1).

- A 4-byte hash function identifier (= 1 for SHA-1, 2 for SHA-256).
- A table of index positions (one per packed object, num_objects in total, each a 4-byte unsigned integer in network order), sorted by their corresponding offsets in the packfile.
- A trailer, containing a:

  ```
  checksum of the corresponding packfile, and

  a checksum of all of the above.
  ```

All 4-byte numbers are in network order.

pack-*.mtimes files have the format:

All 4-byte numbers are in network byte order.

- A 4-byte magic number *0x4d544d45* (*MTME*).
- A 4-byte version identifier (= 1).
- A 4-byte hash function identifier (= 1 for SHA-1, 2 for SHA-256).
- A table of 4-byte unsigned integers. The ith value is the modification time (mtime) of the ith object in the corresponding pack by lexicographic (index) order. The mtimes count standard epoch seconds.
- A trailer, containing a checksum of the corresponding packfile, and a checksum of all of the above (each having length according to the specified hash function).

multi-pack-index (MIDX) files have the following format:

The multi-pack-index files refer to multiple pack-files and loose objects.

In order to allow extensions that add extra data to the MIDX, we organize the body into "chunks" and provide a lookup table at the beginning of the body. The header includes certain length values, such as the number of packs, the number of base MIDX files, hash lengths and types.

All 4-byte numbers are in network order.

HEADER:

```
4-byte signature:
    The signature is: {'M', 'I', 'D', 'X'}

1-byte version number:
    Git only writes or recognizes version 1.

1-byte Object Id Version
    We infer the length of object IDs (OIDs) from this value:
        1 => SHA-1
        2 => SHA-256
    If the hash type does not match the repository's hash algorithm,
    the multi-pack-index file should be ignored with a warning
    presented to the user.

1-byte number of "chunks"

1-byte number of base multi-pack-index files:
    This value is currently always zero.

4-byte number of pack files
```

CHUNK LOOKUP:

```
(C + 1) * 12 bytes providing the chunk offsets:
    First 4 bytes describe chunk id. Value 0 is a terminating label.
    Other 8 bytes provide offset in current file for chunk to start.
    (Chunks are provided in file-order, so you can infer the length
    using the next chunk position if necessary.)

The CHUNK LOOKUP matches the table of contents from
the chunk-based file format, see linkgit:gitformat-chunk[5].

The remaining data in the body is described one chunk at a time, and
these chunks may be given in any order. Chunks are required unless
otherwise specified.
```

CHUNK DATA:

```
Packfile Names (ID: {'P', 'N', 'A', 'M'})
    Store the names of packfiles as a sequence of NUL-terminated
    strings. There is no extra padding between the filenames,
    and they are listed in lexicographic order. The chunk itself
    is padded at the end with between 0 and 3 NUL bytes to make the
    chunk size a multiple of 4 bytes.

Bitmapped Packfiles (ID: {'B', 'T', 'M', 'P'})
    Stores a table of two 4-byte unsigned integers in network order.
    Each table entry corresponds to a single pack (in the order that
    they appear above in the `PNAM` chunk). The values for each table
    entry are as follows:
    - The first bit position (in pseudo-pack order, see below) to
      contain an object from that pack.
    - The number of bits whose objects are selected from that pack.

OID Fanout (ID: {'O', 'I', 'D', 'F'})
    The ith entry, F[i], stores the number of OIDs with first
    byte at most i. Thus F[255] stores the total
    number of objects.

OID Lookup (ID: {'O', 'I', 'D', 'L'})
    The OIDs for all objects in the MIDX are stored in lexicographic
    order in this chunk.

Object Offsets (ID: {'O', 'O', 'F', 'F'})
    Stores two 4-byte values for every object.
    1: The pack-int-id for the pack storing this object.
    2: The offset within the pack.
        If all offsets are less than 2^32, then the large offset chunk
        will not exist and offsets are stored as in IDX v1.
        If there is at least one offset value larger than 2^32-1, then
        the large offset chunk must exist, and offsets larger than
        2^31-1 must be stored in it instead. If the large offset chunk
        exists and the 31st bit is on, then removing that bit reveals
        the row in the large offsets containing the 8-byte offset of
        this object.

[Optional] Object Large Offsets (ID: {'L', 'O', 'F', 'F'})
    8-byte offsets into large packfiles.

[Optional] Bitmap pack order (ID: {'R', 'I', 'D', 'X'})
    A list of MIDX positions (one per object in the MIDX, num_objects in
```

```
total, each a 4-byte unsigned integer in network byte order), sorted
according to their relative bitmap/pseudo-pack positions.
```

TRAILER:

```
Index checksum of the above contents.
```

multi-pack-index reverse indexes

Similar to the pack-based reverse index, the multi-pack index can also be used to generate a reverse index.

Instead of mapping between offset, pack-, and index position, this reverse index maps between an object's position within the MIDX, and that object's position within a pseudo-pack that the MIDX describes (i.e., the ith entry of the multi-pack reverse index holds the MIDX position of ith object in pseudo-pack order).

To clarify the difference between these orderings, consider a multi-pack reachability bitmap (which does not yet exist, but is what we are building towards here). Each bit needs to correspond to an object in the MIDX, and so we need an efficient mapping from bit position to MIDX position.

One solution is to let bits occupy the same position in the oid-sorted index stored by the MIDX. But because oids are effectively random, their resulting reachability bitmaps would have no locality, and thus compress poorly. (This is the reason that single-pack bitmaps use the pack ordering, and not the .idx ordering, for the same purpose.)

So we'd like to define an ordering for the whole MIDX based around pack ordering, which has far better locality (and thus compresses more efficiently). We can think of a pseudo-pack created by the concatenation of all of the packs in the MIDX. E.g., if we had a MIDX with three packs (a, b, c), with 10, 15, and 20 objects respectively, we can imagine an ordering of the objects like:

```
|a,0|a,1|...|a,9|b,0|b,1|...|b,14|c,0|c,1|...|c,19|
```

where the ordering of the packs is defined by the MIDX's pack list, and then the ordering of objects within each pack is the same as the order in the actual packfile.

Given the list of packs and their counts of objects, you can naïvely reconstruct that pseudo-pack ordering (e.g., the object at position 27 must be (c,1) because packs "a" and "b" consumed 25 of the slots). But there's a catch. Objects may be duplicated between packs, in which case the MIDX only stores one pointer to the object (and thus we'd want only one slot in the bitmap).

Callers could handle duplicates themselves by reading objects in order of their bit-position, but that's linear in the number of objects, and much too expensive for ordinary bitmap lookups. Building a reverse index solves this, since it is the logical inverse of the index, and that index has already removed duplicates. But, building a reverse index on the fly can be expensive. Since we already have an on-disk format for pack-based reverse indexes, let's reuse it for the MIDX's pseudo-pack, too.

Objects from the MIDX are ordered as follows to string together the pseudo-pack. Let *pack(o)* return the pack from which *o* was selected by the MIDX, and define an ordering of packs based on their numeric ID (as stored by the MIDX). Let *offset(o)* return the object offset of *o* within *pack(o)*. Then, compare *o1* and *o2* as follows:

- If one of *pack(o1)* and *pack(o2)* is preferred and the other is not, then the preferred one sorts first.

  (This is a detail that allows the MIDX bitmap to determine which pack should be used by the pack-reuse mechanism, since it can ask the MIDX for the pack containing the object at bit position 0).

- If *pack(o1) # pack(o2)*, then sort the two objects in descending order based on the pack ID.

- Otherwise, *pack(o1) = pack(o2)*, and the objects are sorted in pack-order (i.e., *o1* sorts ahead of *o2* exactly when *offset(o1) < offset(o2)*).

In short, a MIDX's pseudo-pack is the de-duplicated concatenation of objects in packs stored by the MIDX, laid out in pack order, and the packs arranged in MIDX order (with the preferred pack coming first).

The MIDX's reverse index is stored in the optional *RIDX* chunk within the MIDX itself.

1. *BTMP* chunk

The Bitmapped Packfiles (*BTMP*) chunk encodes additional information about the objects in the multi-pack index's reachability bitmap. Recall that objects from the MIDX are arranged in "pseudo-pack" order (see above) for reachability bitmaps.

From the example above, suppose we have packs "a", "b", and "c", with 10, 15, and 20 objects, respectively. In pseudo-pack order, those would be arranged as follows:

```
|a,0|a,1|...|a,9|b,0|b,1|...|b,14|c,0|c,1|...|c,19|
```

When working with single-pack bitmaps (or, equivalently, multi-pack reachability bitmaps with a preferred pack), Section G.3.96, “git-pack-objects(1)” performs verbatim reuse, attempting to reuse chunks of the bitmapped or preferred packfile instead of adding objects to the packing list.

When a chunk of bytes is reused from an existing pack, any objects contained therein do not need to be added to the packing list, saving memory and CPU time. But a chunk from an existing packfile can only be reused when the following conditions are met:

- The chunk contains only objects which were requested by the caller (i.e. does not contain any objects which the caller didn't ask for explicitly or implicitly).
- All objects stored in non-thin packs as offset- or reference-deltas also include their base object in the resulting pack.

The *BTMP* chunk encodes the necessary information in order to implement multi-pack reuse over a set of packfiles as described above. Specifically, the *BTMP* chunk encodes three pieces of information (all 32-bit unsigned integers in network byte-order) for each packfile *p* that is stored in the MIDX, as follows:

bitmap_pos

The first bit position (in pseudo-pack order) in the multi-pack index's reachability bitmap occupied by an object from *p*.

bitmap_nr

The number of bit positions (including the one at *bitmap_pos*) that encode objects from that pack *p*.

For example, the *BTMP* chunk corresponding to the above example (with packs a, b, and c) would look like:

	bitmap_pos	*bitmap_nr*
packfile a	*0*	*10*
packfile b	*10*	*15*
packfile c	*25*	*20*

With this information in place, we can treat each packfile as individually reusable in the same fashion as verbatim pack reuse is performed on individual packs prior to the implementation of the *BTMP* chunk.

cruft packs

The cruft packs feature offer an alternative to Git's traditional mechanism of removing unreachable objects. This document provides an overview of Git's pruning mechanism, and how a cruft pack can be used instead to accomplish the same.

1. Background

To remove unreachable objects from your repository, Git offers *git repack -Ad* (see Section G.3.113, “git-repack(1)”). Quoting from the documentation:

```
[...] unreachable objects in a previous pack become loose, unpacked
 objects,
instead of being left in the old pack. [...] loose unreachable objects will
 be
pruned according to normal expiry rules with the next 'git gc' invocation.
```

Unreachable objects aren't removed immediately, since doing so could race with an incoming push which may reference an object which is about to be deleted. Instead, those unreachable objects are stored as loose objects and stay that way until they are older than the expiration window, at which point they are removed by Section G.3.101, “git-prune(1)”.

Git must store these unreachable objects loose in order to keep track of their per-object mtimes. If these unreachable objects were written into one big pack, then either freshening that pack (because an object contained within it was re-written) or creating a new pack of unreachable objects would cause the pack's mtime to get updated, and the objects within it would never leave the expiration window. Instead, objects are stored loose in order to keep track of the individual object mtimes and avoid a situation where all cruft objects are freshened at once.

This can lead to undesirable situations when a repository contains many unreachable objects which have not yet left the grace period. Having large directories in the shards of *.git/objects* can lead to decreased performance in the repository. But given enough unreachable objects, this can lead to inode starvation and degrade the performance of the whole system. Since we can never pack those objects, these repositories often take up a large amount of disk space, since we can only zlib compress them, but not store them in delta chains.

2. Cruft packs

A cruft pack eliminates the need for storing unreachable objects in a loose state by including the per-object mtimes in a separate file alongside a single pack containing all loose objects.

A cruft pack is written by *git repack --cruft* when generating a new pack. Section G.3.96, “git-pack-objects(1)”'s *--cruft* option. Note that *git repack --cruft* is a classic all-into-one repack, meaning that everything in the resulting pack is reachable, and everything else is unreachable. Once written, the *--cruft* option instructs *git repack* to generate another pack containing only objects not packed in the previous step (which equates to packing all unreachable objects together). This progresses as follows:

1. Enumerate every object, marking any object which is (a) not contained in a kept-pack, and (b) whose mtime is within the grace period as a traversal tip.
2. Perform a reachability traversal based on the tips gathered in the previous step, adding every object along the way to the pack.
3. Write the pack out, along with a *.mtimes* file that records the per-object timestamps.

This mode is invoked internally by Section G.3.113, “git-repack(1)” when instructed to write a cruft pack. Crucially, the set of in-core kept packs is exactly the set of packs which will not be deleted by the repack; in other words, they contain all of the repository's reachable objects.

When a repository already has a cruft pack, *git repack --cruft* typically only adds objects to it. An exception to this is when *git repack* is given the *--cruft-expiration* option, which allows the generated cruft pack to omit expired objects instead of waiting for Section G.3.58, “git-gc(1)” to expire those objects later on.

It is Section G.3.58, “git-gc(1)” that is typically responsible for removing expired unreachable objects.

3. Alternatives

Notable alternatives to this design include:

- The location of the per-object mtime data.

On the location of mtime data, a new auxiliary file tied to the pack was chosen to avoid complicating the *.idx* format. If the *.idx* format were ever to gain support for optional chunks of data, it may make sense to consolidate the *.mtimes* format into the *.idx* itself.

GIT

Part of the Section G.3.1, “git(1)” suite

G.5.6. gitformat-signature(5)

2024-07-02
Revision History
2024-07-02

NAME

gitformat-signature - Git cryptographic signature formats

SYNOPSIS

<[tag|commit] object header(s)>
<over-the-wire protocol>

DESCRIPTION

Git uses cryptographic signatures in various places, currently objects (tags, commits, mergetags) and transactions (pushes). In every case, the command which is about to create an object or transaction determines a payload from that, calls an external program to obtain a detached signature for the payload (*gpg -bsa* in the case of PGP signatures), and embeds the signature into the object or transaction.

Signatures begin with an "ASCII Armor" header line and end with a tail line, which differ depending on signature type (as selected by *gpg.format*, see Section G.3.29, “git-config(1)”). These are, for *gpg.format* values:

gpg (PGP)

-----*BEGIN PGP SIGNATURE*----- and -----*END PGP SIGNATURE*-----. Or, if gpg is told to produce RFC1991 signatures, -----*BEGIN PGP MESSAGE*----- and -----*END PGP MESSAGE*-----

ssh (SSH)

-----*BEGIN SSH SIGNATURE*----- and -----*END SSH SIGNATURE*-----

x509 (X.509)

-----*BEGIN SIGNED MESSAGE*----- and -----*END SIGNED MESSAGE*-----

Signatures sometimes appear as a part of the normal payload (e.g. a signed tag has the signature block appended after the payload that the signature applies to), and sometimes appear in the value of an object header (e.g. a merge commit that merged a signed tag would have the entire tag contents on its "mergetag" header). In the case of the latter, the usual multi-line formatting rule for object headers applies. I.e. the second and subsequent lines are prefixed with a SP to signal that the line is continued from the previous line.

This is even true for an originally empty line. In the following examples, the end of line that ends with a whitespace letter is highlighted with a *$* sign; if you are trying to recreate these example by hand, do not cut and paste them--they are there primarily to highlight extra whitespace at the end of some lines.

The signed payload and the way the signature is embedded depends on the type of the object resp. transaction.

Tag signatures

- created by: *git tag -s*
- payload: annotated tag object
- embedding: append the signature to the unsigned tag object

- example: tag *signedtag* with subject *signed tag*

```
object 04b871796dc0420f8e7561a895b52484b701d51a
type commit
tag signedtag
tagger C O Mitter <committer@example.com> 1465981006 +0000

signed tag

signed tag message body
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJXYRhOAAoJEGEJLoW3InGJklkIAIcnhL7RwEb/+QeX9enkXhxn
rxfdqrvWd1K80sl2TOt8Bg/NYwrUBw/RWJ+sg/hhHp4WtvE1HDGHlkEz3y11Lkuh
8tSxS3qKTxXUGozyPGuE90sJfExhZlW4knIQ1wt/yWqM+33E9pN4hzPqLwyrdods
q8FWEqPPUbSJXoMbRPw04S5jrLtZSsUWbRYjmJCHzlhSfFWW4eFd37uquIaLUBS0
rkC3Jrx7420jkIpgFcTI2s60uhSQLzgcCwdA2ukSYIRnjg/zDkj8+3h/GaROJ72x
lZyI6HWixKJkWw8lE9aAOD9TmTW9sFJwcVAzmAuFX2kUreDUKMZduGcoRYGpD7E=
=jpXa
-----END PGP SIGNATURE-----
```

- verify with: *git verify-tag [-v]* or *git tag -v*

```
gpg: Signature made Wed Jun 15 10:56:46 2016 CEST using RSA key ID B7227189
gpg: Good signature from "Eris Discordia <discord@example.net>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the
 owner.
Primary key fingerprint: D4BE 2231 1AD3 131E 5EDA  29A4 6109 2E85 B722 7189
object 04b871796dc0420f8e7561a895b52484b701d51a
type commit
tag signedtag
tagger C O Mitter <committer@example.com> 1465981006 +0000

signed tag

signed tag message body
```

Commit signatures

- created by: *git commit -S*
- payload: commit object
- embedding: header entry *gpgsig* (content is preceded by a space)
- example: commit with subject *signed commit*

```
tree eebfed94e75e7760540d1485c740902590a00332
parent 04b871796dc0420f8e7561a895b52484b701d51a
author A U Thor <author@example.com> 1465981137 +0000
committer C O Mitter <committer@example.com> 1465981137 +0000
gpgsig -----BEGIN PGP SIGNATURE-----
 Version: GnuPG v1
 $
 iQEcBAABAgAGBQJXYRjRAAoJEGEJLoW3InGJ3IwIAIY4SA6GxY3BjL60YyvsJPh/
 HRCJwH+w7wt3Yc/9/bW2F+gF72kdHOOs2jfv+OZhq0q4OAN6fvVSczISY/82LpS7
 DVdMQj2/YcHDT4xrDNBnXnviDO9G7am/9OE77kEbXrp7QPxvhjkicHNwy2rEflAA
```

```
 zn075rtEERDHr8nRYiDh8eVrefSO7D+bdQ7gv+7GsYMsd2auJWi1dHOSfTr9HIF4
 HJhWXT9d2f8W+diRYXGh4X0wYiGg6na/soXc+vdtDYBzIxanRqjg8jCAeo1eOTk1
 EdTwhcTZlI0x5pvJ3H0+4hA2jtldVtmPM4OTB0cTrEWBad7XV6YgiyuII73Ve3I=
 =jKHM
 -----END PGP SIGNATURE-----

signed commit

signed commit message body
```

- verify with: *git verify-commit [-v]* (or *git show --show-signature*)

```
gpg: Signature made Wed Jun 15 10:58:57 2016 CEST using RSA key ID B7227189
gpg: Good signature from "Eris Discordia <discord@example.net>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the
 owner.
Primary key fingerprint: D4BE 2231 1AD3 131E 5EDA  29A4 6109 2E85 B722 7189
tree eebfed94e75e7760540d1485c740902590a00332
parent 04b871796dc0420f8e7561a895b52484b701d51a
author A U Thor <author@example.com> 1465981137 +0000
committer C O Mitter <committer@example.com> 1465981137 +0000

signed commit

signed commit message body
```

Mergetag signatures

- created by: *git merge* on signed tag
- payload/embedding: the whole signed tag object is embedded into the (merge) commit object as header entry *mergetag*
- example: merge of the signed tag *signedtag* as above

```
tree c7b1cff039a93f3600a1d18b82d26688668c7dea
parent c33429be94b5f2d3ee9b0adad223f877f174b05d
parent 04b871796dc0420f8e7561a895b52484b701d51a
author A U Thor <author@example.com> 1465982009 +0000
committer C O Mitter <committer@example.com> 1465982009 +0000
mergetag object 04b871796dc0420f8e7561a895b52484b701d51a
 type commit
 tag signedtag
 tagger C O Mitter <committer@example.com> 1465981006 +0000
 $
 signed tag
 $
 signed tag message body
 -----BEGIN PGP SIGNATURE-----
 Version: GnuPG v1
 $
 iQEcBAABAgAGBQJXYRhOAAoJEGEJLoW3InGJklkIAIcnhL7RwEb/+QeX9enkXhxn
 rxfdqrvWd1K80sl2TOt8Bg/NYwrUBw/RWJ+sg/hhHp4WtvE1HDGHlkEz3y11Lkuh
 8tSxS3qKTxXUGozyPGuE90sJfExhZlW4knIQ1wt/yWqM+33E9pN4hzPqLwyrdods
 q8FWEqPPUbSJXoMbRPw04S5jrLtZSsUWbRYjmJCHzlhSfFWW4eFd37uquIaLUBS0
 rkC3Jrx7420jkIpgFcTI2s60uhSQLzgcCwdA2ukSYIRnjg/zDkj8+3h/GaROJ72x
 lZyI6HWixKJkWw8lE9aAOD9TmTW9sFJwcVAzmAuFX2kUreDUKMZduGcoRYGpD7E=
 =jpXa
```

```
 -----END PGP SIGNATURE-----

Merge tag 'signedtag' into downstream

signed tag

signed tag message body

# gpg: Signature made Wed Jun 15 08:56:46 2016 UTC using RSA key ID
 B7227189
# gpg: Good signature from "Eris Discordia <discord@example.net>"
# gpg: WARNING: This key is not certified with a trusted signature!
# gpg:          There is no indication that the signature belongs to the
 owner.
# Primary key fingerprint: D4BE 2231 1AD3 131E 5EDA  29A4 6109 2E85 B722
 7189
```

- verify with: verification is embedded in merge commit message by default, alternatively with *git show --show-signature*:

```
commit 9863f0c76ff78712b6800e199a46aa56afbcbd49
merged tag 'signedtag'
gpg: Signature made Wed Jun 15 10:56:46 2016 CEST using RSA key ID B7227189
gpg: Good signature from "Eris Discordia <discord@example.net>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the
 owner.
Primary key fingerprint: D4BE 2231 1AD3 131E 5EDA  29A4 6109 2E85 B722 7189
Merge: c33429b 04b8717
Author: A U Thor <author@example.com>
Date:   Wed Jun 15 09:13:29 2016 +0000

    Merge tag 'signedtag' into downstream

    signed tag

    signed tag message body

    # gpg: Signature made Wed Jun 15 08:56:46 2016 UTC using RSA key ID
 B7227189
    # gpg: Good signature from "Eris Discordia <discord@example.net>"
    # gpg: WARNING: This key is not certified with a trusted signature!
    # gpg:          There is no indication that the signature belongs to
 the owner.
    # Primary key fingerprint: D4BE 2231 1AD3 131E 5EDA  29A4 6109 2E85
 B722 7189
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.5.7. gitprotocol-capabilities(5)

2024-07-02

Revision History

2024-07-02

NAME

gitprotocol-capabilities - Protocol v0 and v1 capabilities

SYNOPSIS

<over-the-wire-protocol>

DESCRIPTION

> **Note**
>
> this document describes capabilities for versions 0 and 1 of the pack protocol. For version 2, please refer to the Section G.5.11, “gitprotocol-v2(5)” doc.

Servers SHOULD support all capabilities defined in this document.

On the very first line of the initial server response of either receive-pack and upload-pack the first reference is followed by a NUL byte and then a list of space delimited server capabilities. These allow the server to declare what it can and cannot support to the client.

Client will then send a space separated list of capabilities it wants to be in effect. The client MUST NOT ask for capabilities the server did not say it supports.

Server MUST diagnose and abort if capabilities it does not understand were sent. Server MUST NOT ignore capabilities that client requested and server advertised. As a consequence of these rules, server MUST NOT advertise capabilities it does not understand.

The *atomic*, *report-status*, *report-status-v2*, *delete-refs*, *quiet*, and *push-cert* capabilities are sent and recognized by the receive-pack (push to server) process.

The *ofs-delta* and *side-band-64k* capabilities are sent and recognized by both upload-pack and receive-pack protocols. The *agent* and *session-id* capabilities may optionally be sent in both protocols.

All other capabilities are only recognized by the upload-pack (fetch from server) process.

multi_ack

The *multi_ack* capability allows the server to return "ACK obj-id continue" as soon as it finds a commit that it can use as a common base, between the client's wants and the client's have set.

By sending this early, the server can potentially head off the client from walking any further down that particular branch of the client's repository history. The client may still need to walk down other branches, sending have lines for those, until the server has a complete cut across the DAG, or the client has said "done".

Without multi_ack, a client sends have lines in --date-order until the server has found a common base. That means the client will send have lines that are already known by the server to be common, because they overlap in time with another branch on which the server hasn't found a common base yet.

For example suppose the client has commits in caps that the server doesn't and the server has commits in lower case that the client doesn't, as in the following diagram:

```
   +---- u ---------------------- x
  /              +----- y
 /              /
a -- b -- c -- d -- E -- F
   \
    +--- Q -- R -- S
```

If the client wants x,y and starts out by saying have F,S, the server doesn't know what F,S is. Eventually the client says "have d" and the server sends "ACK d continue" to let the client know to stop walking down that line (so don't send c-b-a), but it's not done yet, it needs a base for x. The client keeps going with S-R-Q, until a gets reached, at which point the server has a clear base and it all ends.

Without multi_ack the client would have sent that c-b-a chain anyway, interleaved with S-R-Q.

multi_ack_detailed

This is an extension of multi_ack that permits the client to better understand the server's in-memory state. See Section G.5.10, “gitprotocol-pack(5)”, section "Packfile Negotiation" for more information.

no-done

This capability should only be used with the smart HTTP protocol. If multi_ack_detailed and no-done are both present, then the sender is free to immediately send a pack following its first "ACK obj-id ready" message.

Without no-done in the smart HTTP protocol, the server session would end and the client has to make another trip to send "done" before the server can send the pack. no-done removes the last round and thus slightly reduces latency.

thin-pack

A thin pack is one with deltas which reference base objects not contained within the pack (but are known to exist at the receiving end). This can reduce the network traffic significantly, but it requires the receiving end to know how to "thicken" these packs by adding the missing bases to the pack.

The upload-pack server advertises *thin-pack* when it can generate and send a thin pack. A client requests the *thin-pack* capability when it understands how to "thicken" it, notifying the server that it can receive such a pack. A client MUST NOT request the *thin-pack* capability if it cannot turn a thin pack into a self-contained pack.

Receive-pack, on the other hand, is assumed by default to be able to handle thin packs, but can ask the client not to use the feature by advertising the *no-thin* capability. A client MUST NOT send a thin pack if the server advertises the *no-thin* capability.

The reasons for this asymmetry are historical. The receive-pack program did not exist until after the invention of thin packs, so historically the reference implementation of receive-pack always understood thin packs. Adding *no-thin* later allowed receive-pack to disable the feature in a backwards-compatible manner.

side-band, side-band-64k

This capability means that the server can send, and the client can understand, multiplexed progress reports and error info interleaved with the packfile itself.

These two options are mutually exclusive. A modern client always favors *side-band-64k*.

Either mode indicates that the packfile data will be streamed broken up into packets of up to either 1000 bytes in the case of *side_band*, or 65520 bytes in the case of *side_band_64k*. Each packet is made up of a leading 4-byte pkt-line length of how much data is in the packet, followed by a 1-byte stream code, followed by the actual data.

The stream code can be one of:

```
1 - pack data
2 - progress messages
3 - fatal error message just before stream aborts
```

The "side-band-64k" capability came about as a way for newer clients that can handle much larger packets to request packets that are actually crammed nearly full, while maintaining backward compatibility for the older clients.

Further, with side-band and its up to 1000-byte messages, it's actually 999 bytes of payload and 1 byte for the stream code. With side-band-64k, same deal, you have up to 65519 bytes of data and 1 byte for the stream code.

The client MUST send only one of "side-band" and "side- band-64k". The server MUST diagnose it as an error if client requests both.

ofs-delta

The server can send, and the client can understand, PACKv2 with delta referring to its base by position in pack rather than by an obj-id. That is, they can send/read OBJ_OFS_DELTA (aka type 6) in a packfile.

agent

The server may optionally send a capability of the form *agent=X* to notify the client that the server is running version *X*. The client may optionally return its own agent string by responding with an *agent=Y* capability (but it MUST NOT do so if the server did not mention the agent capability). The *X* and *Y* strings may contain any printable ASCII characters except space (i.e., the byte range 32 < x < 127), and are typically of the form "package/version" (e.g., "git/1.8.3.1"). The agent strings are purely informative for statistics and debugging purposes, and MUST NOT be used to programmatically assume the presence or absence of particular features.

object-format

This capability, which takes a hash algorithm as an argument, indicates that the server supports the given hash algorithms. It may be sent multiple times; if so, the first one given is the one used in the ref advertisement.

When provided by the client, this indicates that it intends to use the given hash algorithm to communicate. The algorithm provided must be one that the server supports.

If this capability is not provided, it is assumed that the only supported algorithm is SHA-1.

symref

This parameterized capability is used to inform the receiver which symbolic ref points to which ref; for example, "symref=HEAD:refs/heads/master" tells the receiver that HEAD points to master. This capability can be repeated to represent multiple symrefs.

Servers SHOULD include this capability for the HEAD symref if it is one of the refs being sent.

Clients MAY use the parameters from this capability to select the proper initial branch when cloning a repository.

shallow

This capability adds "deepen", "shallow" and "unshallow" commands to the fetch-pack/upload-pack protocol so clients can request shallow clones.

deepen-since

This capability adds "deepen-since" command to fetch-pack/upload-pack protocol so the client can request shallow clones that are cut at a specific time, instead of depth. Internally it's equivalent of doing "rev-list --max-age=<timestamp>" on the server side. "deepen-since" cannot be used with "deepen".

deepen-not

This capability adds "deepen-not" command to fetch-pack/upload-pack protocol so the client can request shallow clones that are cut at a specific revision, instead of depth. Internally it's equivalent of doing "rev-list --not <rev>" on the server side. "deepen-not" cannot be used with "deepen", but can be used with "deepen-since".

deepen-relative

If this capability is requested by the client, the semantics of "deepen" command is changed. The "depth" argument is the depth from the current shallow boundary, instead of the depth from remote refs.

no-progress

The client was started with "git clone -q" or something similar, and doesn't want that side band 2. Basically the client just says "I do not wish to receive stream 2 on sideband, so do not send it to me, and if you did, I will drop it on the floor anyway". However, the sideband channel 3 is still used for error responses.

include-tag

The *include-tag* capability is about sending annotated tags if we are sending objects they point to. If we pack an object to the client, and a tag object points exactly at that object, we pack the tag object too. In general this allows a client to get all new annotated tags when it fetches a branch, in a single network connection.

Clients MAY always send include-tag, hardcoding it into a request when the server advertises this capability. The decision for a client to request include-tag only has to do with the client's desires for tag data, whether or not a server had advertised objects in the refs/tags/* namespace.

Servers MUST pack the tags if their referent is packed and the client has requested include-tags.

Clients MUST be prepared for the case where a server has ignored include-tag and has not actually sent tags in the pack. In such cases the client SHOULD issue a subsequent fetch to acquire the tags that include-tag would have otherwise given the client.

The server SHOULD send include-tag, if it supports it, regardless of whether or not there are tags available.

report-status

The receive-pack process can receive a *report-status* capability, which tells it that the client wants a report of what happened after a packfile upload and reference update. If the pushing client requests this capability, after unpacking and updating references the server will respond with whether the packfile unpacked successfully and if each reference was updated successfully. If any of those were not successful, it will send back an error message. See Section G.5.10, “gitprotocol-pack(5)” for example messages.

report-status-v2

Capability *report-status-v2* extends capability *report-status* by adding new "option" directives in order to support reference rewritten by the "proc-receive" hook. The "proc-receive" hook may handle a command for a pseudo-reference which may create or update a reference with different name, new-oid, and old-oid. While the capability *report-status* cannot report for such case. See Section G.5.10, “gitprotocol-pack(5)” for details.

delete-refs

If the server sends back the *delete-refs* capability, it means that it is capable of accepting a zero-id value as the target value of a reference update. It is not sent back by the client, it simply informs the client that it can be sent zero-id values to delete references.

quiet

If the receive-pack server advertises the *quiet* capability, it is capable of silencing human-readable progress output which otherwise may be shown when processing the received pack. A send-pack client should respond with the *quiet* capability to suppress server-side progress reporting if the local progress reporting is also being suppressed (e.g., via *push -q*, or if stderr does not go to a tty).

atomic

If the server sends the *atomic* capability it is capable of accepting atomic pushes. If the pushing client requests this capability, the server will update the refs in one atomic transaction. Either all refs are updated or none.

push-options

If the server sends the *push-options* capability it is able to accept push options after the update commands have been sent, but before the packfile is streamed. If the pushing client requests this capability, the server will pass the options to the pre- and post- receive hooks that process this push request.

allow-tip-sha1-in-want

If the upload-pack server advertises this capability, fetch-pack may send "want" lines with object names that exist at the server but are not advertised by upload-pack. For historical reasons, the name of this capability contains "sha1". Object names are always given using the object format negotiated through the *object-format* capability.

allow-reachable-sha1-in-want

If the upload-pack server advertises this capability, fetch-pack may send "want" lines with object names that exist at the server but are not advertised by upload-pack. For historical reasons, the name of this capability contains "sha1". Object names are always given using the object format negotiated through the *object-format* capability.

push-cert=<nonce>

The receive-pack server that advertises this capability is willing to accept a signed push certificate, and asks the <nonce> to be included in the push certificate. A send-pack client MUST NOT send a push-cert packet unless the receive-pack server advertises this capability.

filter

If the upload-pack server advertises the *filter* capability, fetch-pack may send "filter" commands to request a partial clone or partial fetch and request that the server omit various objects from the packfile.

session-id=<session-id>

The server may advertise a session ID that can be used to identify this process across multiple requests. The client may advertise its own session ID back to the server as well.

Session IDs should be unique to a given process. They must fit within a packet-line, and must not contain non-printable or whitespace characters. The current implementation uses trace2 session IDs (see *api-trace2* [https://www.kernel.org/pub/software/scm/git/docs/technical/api-trace2.html] for details), but this may change and users of the session ID should not rely on this fact.

GIT

Part of the Section G.3.1, “git(1)” suite

G.5.8. gitprotocol-common(5)

2024-07-02

Revision History

2024-07-02

NAME

gitprotocol-common - Things common to various protocols

SYNOPSIS

<over-the-wire-protocol>

DESCRIPTION

This document defines things common to various over-the-wire protocols and file formats used in Git.

ABNF Notation

ABNF notation as described by RFC 5234 is used within the protocol documents, except the following replacement core rules are used:

```
HEXDIG    =  DIGIT / "a" / "b" / "c" / "d" / "e" / "f"
```

We also define the following common rules:

```
NUL       =  %x00
zero-id   =  40*"0"
obj-id    =  40*(HEXDIGIT)
```

```
refname  =  "HEAD"
refname /=  "refs/" <see discussion below>
```

A refname is a hierarchical octet string beginning with "refs/" and not violating the *git-check-ref-format* command's validation rules. More specifically, they:

1. They can include slash / for hierarchical (directory) grouping, but no slash-separated component can begin with a dot ..
2. They must contain at least one /. This enforces the presence of a category like *heads/*, *tags/* etc. but the actual names are not restricted.
3. They cannot have two consecutive dots .. anywhere.
4. They cannot have ASCII control characters (i.e. bytes whose values are lower than \040, or \177 *DEL*), space, tilde ~, caret ^, colon *:*, question-mark *?*, asterisk *, or open bracket *[* anywhere.
5. They cannot end with a slash / or a dot ..
6. They cannot end with the sequence *.lock*.
7. They cannot contain a sequence *@{*.
8. They cannot contain a \\.

pkt-line Format

Much (but not all) of the payload is described around pkt-lines.

A pkt-line is a variable length binary string. The first four bytes of the line, the pkt-len, indicates the total length of the line, in hexadecimal. The pkt-len includes the 4 bytes used to contain the length's hexadecimal representation.

A pkt-line MAY contain binary data, so implementors MUST ensure pkt-line parsing/formatting routines are 8-bit clean.

A non-binary line SHOULD BE terminated by an LF, which if present MUST be included in the total length. Receivers MUST treat pkt-lines with non-binary data the same whether or not they contain the trailing LF (stripping the LF if present, and not complaining when it is missing).

The maximum length of a pkt-line's data component is 65516 bytes. Implementations MUST NOT send pkt-line whose length exceeds 65520 (65516 bytes of payload + 4 bytes of length data).

Implementations SHOULD NOT send an empty pkt-line ("0004").

A pkt-line with a length field of 0 ("0000"), called a flush-pkt, is a special case and MUST be handled differently than an empty pkt-line ("0004").

```
pkt-line     =  data-pkt / flush-pkt

data-pkt     =  pkt-len pkt-payload
pkt-len      =  4*(HEXDIG)
pkt-payload  =  (pkt-len - 4)*(OCTET)

flush-pkt    = "0000"
```

Examples (as C-style strings):

```
pkt-line          actual value
---------------------------------
"0006a\n"         "a\n"
```

```
"0005a"           "a"
"000bfoobar\n"    "foobar\n"
"0004"            ""
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.5.9. gitprotocol-http(5)

2024-07-02

Revision History

2024-07-02

NAME

gitprotocol-http - Git HTTP-based protocols

SYNOPSIS

<over-the-wire-protocol>

DESCRIPTION

Git supports two HTTP based transfer protocols. A "dumb" protocol which requires only a standard HTTP server on the server end of the connection, and a "smart" protocol which requires a Git aware CGI (or server module). This document describes both protocols.

As a design feature smart clients can automatically upgrade "dumb" protocol URLs to smart URLs. This permits all users to have the same published URL, and the peers automatically select the most efficient transport available to them.

URL Format

URLs for Git repositories accessed by HTTP use the standard HTTP URL syntax documented by RFC 1738, so they are of the form:

```
http://<host>:<port>/<path>?<searchpart>
```

Within this documentation the placeholder *$GIT_URL* will stand for the http:// repository URL entered by the end-user.

Servers SHOULD handle all requests to locations matching *$GIT_URL*, as both the "smart" and "dumb" HTTP protocols used by Git operate by appending additional path components onto the end of the user supplied *$GIT_URL* string.

An example of a dumb client requesting a loose object:

```
$GIT_URL:     http://example.com:8080/git/repo.git
URL request:  http://example.com:8080/git/repo.git/objects/
d0/49f6c27a2244e12041955e262a404c7faba355
```

An example of a smart request to a catch-all gateway:

```
$GIT_URL:     http://example.com/daemon.cgi?svc=git&q=
URL request:  http://example.com/daemon.cgi?svc=git&q=/info/
refs&service=git-receive-pack
```

An example of a request to a submodule:

```
$GIT_URL:     http://example.com/git/repo.git/path/submodule.git
URL request:  http://example.com/git/repo.git/path/submodule.git/info/refs
```

Clients MUST strip a trailing */*, if present, from the user supplied *$GIT_URL* string to prevent empty path tokens (*//*) from appearing in any URL sent to a server. Compatible clients MUST expand *$GIT_URL/info/refs* as *foo/info/refs* and not *foo//info/refs*.

Authentication

Standard HTTP authentication is used if authentication is required to access a repository, and MAY be configured and enforced by the HTTP server software.

Because Git repositories are accessed by standard path components server administrators MAY use directory based permissions within their HTTP server to control repository access.

Clients SHOULD support Basic authentication as described by RFC 2617. Servers SHOULD support Basic authentication by relying upon the HTTP server placed in front of the Git server software.

Servers SHOULD NOT require HTTP cookies for the purposes of authentication or access control.

Clients and servers MAY support other common forms of HTTP based authentication, such as Digest authentication.

SSL

Clients and servers SHOULD support SSL, particularly to protect passwords when relying on Basic HTTP authentication.

Session State

The Git over HTTP protocol (much like HTTP itself) is stateless from the perspective of the HTTP server side. All state MUST be retained and managed by the client process. This permits simple round-robin load-balancing on the server side, without needing to worry about state management.

Clients MUST NOT require state management on the server side in order to function correctly.

Servers MUST NOT require HTTP cookies in order to function correctly. Clients MAY store and forward HTTP cookies during request processing as described by RFC 2616 (HTTP/1.1). Servers SHOULD ignore any cookies sent by a client.

General Request Processing

Except where noted, all standard HTTP behavior SHOULD be assumed by both client and server. This includes (but is not necessarily limited to):

If there is no repository at *$GIT_URL*, or the resource pointed to by a location matching *$GIT_URL* does not exist, the server MUST NOT respond with *200 OK* response. A server SHOULD respond with *404 Not Found*, *410 Gone*, or any other suitable HTTP status code which does not imply the resource exists as requested.

If there is a repository at *$GIT_URL*, but access is not currently permitted, the server MUST respond with the *403 Forbidden* HTTP status code.

Servers SHOULD support both HTTP 1.0 and HTTP 1.1. Servers SHOULD support chunked encoding for both request and response bodies.

Clients SHOULD support both HTTP 1.0 and HTTP 1.1. Clients SHOULD support chunked encoding for both request and response bodies.

Servers MAY return ETag and/or Last-Modified headers.

Clients MAY revalidate cached entities by including If-Modified-Since and/or If-None-Match request headers.

Servers MAY return *304 Not Modified* if the relevant headers appear in the request and the entity has not changed. Clients MUST treat *304 Not Modified* identical to *200 OK* by reusing the cached entity.

Clients MAY reuse a cached entity without revalidation if the Cache-Control and/or Expires header permits caching. Clients and servers MUST follow RFC 2616 for cache controls.

Discovering References

All HTTP clients MUST begin either a fetch or a push exchange by discovering the references available on the remote repository.

1. Dumb Clients

HTTP clients that only support the "dumb" protocol MUST discover references by making a request for the special info/refs file of the repository.

Dumb HTTP clients MUST make a *GET* request to *$GIT_URL/info/refs*, without any search/query parameters.

```
C: GET $GIT_URL/info/refs HTTP/1.0

S: 200 OK
S:
S: 95dcfa3633004da0049d3d0fa03f80589cbcaf31  refs/heads/maint
S: d049f6c27a2244e12041955e262a404c7faba355  refs/heads/master
S: 2cb58b79488a98d2721cea644875a8dd0026b115  refs/tags/v1.0
S: a3c2e2402b99163d1d59756e5f207ae21cccba4c  refs/tags/v1.0^{}
```

The Content-Type of the returned info/refs entity SHOULD be *text/plain; charset=utf-8*, but MAY be any content type. Clients MUST NOT attempt to validate the returned Content-Type. Dumb servers MUST NOT return a return type starting with *application/x-git-*.

Cache-Control headers MAY be returned to disable caching of the returned entity.

When examining the response clients SHOULD only examine the HTTP status code. Valid responses are *200 OK*, or *304 Not Modified*.

The returned content is a UNIX formatted text file describing each ref and its known value. The file SHOULD be sorted by name according to the C locale ordering. The file SHOULD NOT include the default ref named *HEAD*.

```
info_refs   =  *( ref_record )
ref_record  =  any_ref / peeled_ref

any_ref     =  obj-id HTAB refname LF
peeled_ref  =  obj-id HTAB refname LF
               obj-id HTAB refname "^{}" LF
```

2. Smart Clients

HTTP clients that support the "smart" protocol (or both the "smart" and "dumb" protocols) MUST discover references by making a parameterized request for the info/refs file of the repository.

The request MUST contain exactly one query parameter, *service=$servicename*, where *$servicename* MUST be the service name the client wishes to contact to complete the operation. The request MUST NOT contain additional query parameters.

```
C: GET $GIT_URL/info/refs?service=git-upload-pack HTTP/1.0
```

dumb server reply:

```
S: 200 OK
S:
S: 95dcfa3633004da0049d3d0fa03f80589cbcaf31  refs/heads/maint
S: d049f6c27a2244e12041955e262a404c7faba355  refs/heads/master
```

```
S: 2cb58b79488a98d2721cea644875a8dd0026b115  refs/tags/v1.0
S: a3c2e2402b99163d1d59756e5f207ae21cccba4c  refs/tags/v1.0^{}
```

smart server reply:

```
S: 200 OK
S: Content-Type: application/x-git-upload-pack-advertisement
S: Cache-Control: no-cache
S:
S: 001e# service=git-upload-pack\n
S: 0000
S: 004895dcfa3633004da0049d3d0fa03f80589cbcaf31 refs/heads/maint\0multi_ack
\n
S: 003fd049f6c27a2244e12041955e262a404c7faba355 refs/heads/master\n
S: 003c2cb58b79488a98d2721cea644875a8dd0026b115 refs/tags/v1.0\n
S: 003fa3c2e2402b99163d1d59756e5f207ae21cccba4c refs/tags/v1.0^{}\n
S: 0000
```

The client may send Extra Parameters (see Section G.5.10, “gitprotocol-pack(5)”) as a colon-separated string in the Git-Protocol HTTP header.

Uses the *--http-backend-info-refs* option to Section G.3.151, “git-upload-pack(1)”.

2.1. Dumb Server Response

Dumb servers MUST respond with the dumb server reply format.

See the prior section under dumb clients for a more detailed description of the dumb server response.

2.2. Smart Server Response

If the server does not recognize the requested service name, or the requested service name has been disabled by the server administrator, the server MUST respond with the *403 Forbidden* HTTP status code.

Otherwise, smart servers MUST respond with the smart server reply format for the requested service name.

Cache-Control headers SHOULD be used to disable caching of the returned entity.

The Content-Type MUST be *application/x-$servicename-advertisement*. Clients SHOULD fall back to the dumb protocol if another content type is returned. When falling back to the dumb protocol clients SHOULD NOT make an additional request to *$GIT_URL/info/refs*, but instead SHOULD use the response already in hand. Clients MUST NOT continue if they do not support the dumb protocol.

Clients MUST validate the status code is either *200 OK* or *304 Not Modified*.

Clients MUST validate the first five bytes of the response entity matches the regex *^[0-9a-f]{4}#*. If this test fails, clients MUST NOT continue.

Clients MUST parse the entire response as a sequence of pkt-line records.

Clients MUST verify the first pkt-line is *# service=$servicename*. Servers MUST set $servicename to be the request parameter value. Servers SHOULD include an LF at the end of this line. Clients MUST ignore an LF at the end of the line.

Servers MUST terminate the response with the magic *0000* end pkt-line marker.

The returned response is a pkt-line stream describing each ref and its known value. The stream SHOULD be sorted by name according to the C locale ordering. The stream SHOULD include the default ref named *HEAD* as the first ref. The stream MUST include capability declarations behind a NUL on the first ref.

The returned response contains "version 1" if "version=1" was sent as an Extra Parameter.

```
smart_reply     =  PKT-LINE("# service=$servicename" LF)
                   "0000"
                   *1("version 1")
                   ref_list
                   "0000"
ref_list        =  empty_list / non_empty_list

empty_list      =  PKT-LINE(zero-id SP "capabilities^{}" NUL cap-list LF)

non_empty_list  =  PKT-LINE(obj-id SP name NUL cap_list LF)
                   *ref_record

cap-list        =  capability *(SP capability)
capability      =  1*(LC_ALPHA / DIGIT / "-" / "_")
LC_ALPHA        =  %x61-7A

ref_record      =  any_ref / peeled_ref
any_ref         =  PKT-LINE(obj-id SP name LF)
peeled_ref      =  PKT-LINE(obj-id SP name LF)
                   PKT-LINE(obj-id SP name "^{}" LF
```

Smart Service git-upload-pack

This service reads from the repository pointed to by *$GIT_URL*.

Clients MUST first perform ref discovery with *$GIT_URL/info/refs?service=git-upload-pack*.

```
C: POST $GIT_URL/git-upload-pack HTTP/1.0
C: Content-Type: application/x-git-upload-pack-request
C:
C: 0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7\n
C: 0032have 441b40d833fdfa93eb2908e52742248faf0ee993\n
C: 0000

S: 200 OK
S: Content-Type: application/x-git-upload-pack-result
S: Cache-Control: no-cache
S:
S: ....ACK %s, continue
S: ....NAK
```

Clients MUST NOT reuse or revalidate a cached response. Servers MUST include sufficient Cache-Control headers to prevent caching of the response.

Servers SHOULD support all capabilities defined here.

Clients MUST send at least one "want" command in the request body. Clients MUST NOT reference an id in a "want" command which did not appear in the response obtained through ref discovery unless the server advertises capability *allow-tip-sha1-in-want* or *allow-reachable-sha1-in-want*.

```
compute_request   =  want_list
                     have_list
                     request_end
request_end       =  "0000" / "done"

want_list         =  PKT-LINE(want SP cap_list LF)
                     *(want_pkt)
want_pkt          =  PKT-LINE(want LF)
want              =  "want" SP id
cap_list          =  capability *(SP capability)
```

```
have_list         =  *PKT-LINE("have" SP id LF)
```

TODO: Document this further.

1. The Negotiation Algorithm

The computation to select the minimal pack proceeds as follows (C = client, S = server):

init step:

C: Use ref discovery to obtain the advertised refs.

C: Place any object seen into set *advertised*.

C: Build an empty set, *common*, to hold the objects that are later determined to be on both ends.

C: Build a set, *want*, of the objects from *advertised* that the client wants to fetch, based on what it saw during ref discovery.

C: Start a queue, *c_pending*, ordered by commit time (popping newest first). Add all client refs. When a commit is popped from the queue its parents SHOULD be automatically inserted back. Commits MUST only enter the queue once.

one compute step:

C: Send one *$GIT_URL/git-upload-pack* request:

```
C: 0032want <want-#1>...............................
C: 0032want <want-#2>...............................
....
C: 0032have <common-#1>.............................
C: 0032have <common-#2>.............................
....
C: 0032have <have-#1>...............................
C: 0032have <have-#2>...............................
....
C: 0000
```

The stream is organized into "commands", with each command appearing by itself in a pkt-line. Within a command line, the text leading up to the first space is the command name, and the remainder of the line to the first LF is the value. Command lines are terminated with an LF as the last byte of the pkt-line value.

Commands MUST appear in the following order, if they appear at all in the request stream:

- "want"
- "have"

The stream is terminated by a pkt-line flush (*0000*).

A single "want" or "have" command MUST have one hex formatted object name as its value. Multiple object names MUST be sent by sending multiple commands. Object names MUST be given using the object format negotiated through the *object-format* capability (default SHA-1).

The *have* list is created by popping the first 32 commits from *c_pending*. Fewer can be supplied if *c_pending* empties.

If the client has sent 256 "have" commits and has not yet received one of those back from *s_common*, or the client has emptied *c_pending* it SHOULD include a "done" command to let the server know it won't proceed:

```
C: 0009done
```

S: Parse the git-upload-pack request:

Verify all objects in *want* are directly reachable from refs.

The server MAY walk backwards through history or through the reflog to permit slightly stale requests.

If no "want" objects are received, send an error: TODO: Define error if no "want" lines are requested.

If any "want" object is not reachable, send an error: TODO: Define error if an invalid "want" is requested.

Create an empty list, *s_common*.

If "have" was sent:

Loop through the objects in the order supplied by the client.

For each object, if the server has the object reachable from a ref, add it to *s_common*. If a commit is added to *s_common*, do not add any ancestors, even if they also appear in *have*.

S: Send the git-upload-pack response:

If the server has found a closed set of objects to pack or the request ends with "done", it replies with the pack. TODO: Document the pack based response

```
S: PACK...
```

The returned stream is the side-band-64k protocol supported by the git-upload-pack service, and the pack is embedded into stream 1. Progress messages from the server side MAY appear in stream 2.

Here a "closed set of objects" is defined to have at least one path from every "want" to at least one "common" object.

If the server needs more information, it replies with a status continue response: TODO: Document the non-pack response

C: Parse the upload-pack response: TODO: Document parsing response

Do another compute step.

Smart Service git-receive-pack

This service reads from the repository pointed to by *$GIT_URL*.

Clients MUST first perform ref discovery with *$GIT_URL/info/refs?service=git-receive-pack*.

```
C: POST $GIT_URL/git-receive-pack HTTP/1.0
C: Content-Type: application/x-git-receive-pack-request
C:
C: ....0a53e9ddeaddad63ad106860237bbf53411d11a7
 441b40d833fdfa93eb2908e52742248faf0ee993 refs/heads/maint\0 report-status
C: 0000
C: PACK....
```

```
S: 200 OK
S: Content-Type: application/x-git-receive-pack-result
S: Cache-Control: no-cache
S:
S: ....
```

Clients MUST NOT reuse or revalidate a cached response. Servers MUST include sufficient Cache-Control headers to prevent caching of the response.

Servers SHOULD support all capabilities defined here.

Clients MUST send at least one command in the request body. Within the command portion of the request body clients SHOULD send the id obtained through ref discovery as old_id.

```
update_request  =  command_list
                   "PACK" <binary-data>

command_list    =  PKT-LINE(command NUL cap_list LF)
                   *(command_pkt)
command_pkt     =  PKT-LINE(command LF)
cap_list        =  *(SP capability) SP

command         =  create / delete / update
create          =  zero-id SP new_id SP name
delete          =  old_id SP zero-id SP name
update          =  old_id SP new_id SP name
```

TODO: Document this further.

REFERENCES

RFC 1738: Uniform Resource Locators (URL) [https://www.ietf.org/rfc/rfc1738.txt] RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1 [https://www.ietf.org/rfc/rfc2616.txt]

SEE ALSO

Section G.5.10, “gitprotocol-pack(5)” Section G.5.7, “gitprotocol-capabilities(5)”

GIT

Part of the Section G.3.1, “git(1)” suite

G.5.10. gitprotocol-pack(5)

2024-07-02

Revision History

2024-07-02

NAME

gitprotocol-pack - How packs are transferred over-the-wire

SYNOPSIS

<over-the-wire-protocol>

DESCRIPTION

Git supports transferring data in packfiles over the ssh://, git://, http:// and file:// transports. There exist two sets of protocols, one for pushing data from a client to a server and another for fetching data from a server to a client. The three transports (ssh, git, file) use the same protocol to transfer data. http is documented in Section G.5.9, “gitprotocol-http(5)”.

The processes invoked in the canonical Git implementation are *upload-pack* on the server side and *fetch-pack* on the client side for fetching data; then *receive-pack* on the server and *send-pack* on the client for pushing data. The protocol functions to have a server tell a client what is currently on the server, then for the two to negotiate the smallest amount of data to send in order to fully update one or the other.

pkt-line Format

The descriptions below build on the pkt-line format described in Section G.5.8, “gitprotocol-common(5)”. When the grammar indicates *PKT-LINE(...)*, unless otherwise noted the usual pkt-line LF rules apply: the sender SHOULD include a LF, but the receiver MUST NOT complain if it is not present.

An error packet is a special pkt-line that contains an error string.

```
error-line     =  PKT-LINE("ERR" SP explanation-text)
```

Throughout the protocol, where *PKT-LINE(...)* is expected, an error packet MAY be sent. Once this packet is sent by a client or a server, the data transfer process defined in this protocol is terminated.

Transports

There are three transports over which the packfile protocol is initiated. The Git transport is a simple, unauthenticated server that takes the command (almost always *upload-pack*, though Git servers can be configured to be globally writable, in which *receive- pack* initiation is also allowed) with which the client wishes to communicate and executes it and connects it to the requesting process.

In the SSH transport, the client just runs the *upload-pack* or *receive-pack* process on the server over the SSH protocol and then communicates with that invoked process over the SSH connection.

The file:// transport runs the *upload-pack* or *receive-pack* process locally and communicates with it over a pipe.

Extra Parameters

The protocol provides a mechanism in which clients can send additional information in its first message to the server. These are called "Extra Parameters", and are supported by the Git, SSH, and HTTP protocols.

Each Extra Parameter takes the form of *<key>=<value>* or *<key>*.

Servers that receive any such Extra Parameters MUST ignore all unrecognized keys. Currently, the only Extra Parameter recognized is "version" with a value of *1* or *2*. See Section G.5.11, “gitprotocol-v2(5)” for more information on protocol version 2.

Git Transport

The Git transport starts off by sending the command and repository on the wire using the pkt-line format, followed by a NUL byte and a hostname parameter, terminated by a NUL byte.

```
0033git-upload-pack /project.git\0host=myserver.com\0
```

The transport may send Extra Parameters by adding an additional NUL byte, and then adding one or more NUL-terminated strings:

```
003egit-upload-pack /project.git\0host=myserver.com\0\0version=1\0

git-proto-request = request-command SP pathname NUL
                    [ host-parameter NUL ] [ NUL extra-parameters ]
request-command   = "git-upload-pack" / "git-receive-pack" /
                    "git-upload-archive"   ; case sensitive
pathname          = *( %x01-ff ) ; exclude NUL
host-parameter    = "host=" hostname [ ":" port ]
extra-parameters  = 1*extra-parameter
extra-parameter   = 1*( %x01-ff ) NUL
```

host-parameter is used for the git-daemon name based virtual hosting. See --interpolated-path option to git daemon, with the %H/%CH format characters.

Basically what the Git client is doing to connect to an *upload-pack* process on the server side over the Git protocol is this:

```
$ echo -e -n \
  "003agit-upload-pack /schacon/gitbook.git\0host=example.com\0" |
  nc -v example.com 9418
```

SSH Transport

Initiating the upload-pack or receive-pack processes over SSH is executing the binary on the server via SSH remote execution. It is basically equivalent to running this:

```
$ ssh git.example.com "git-upload-pack '/project.git'"
```

For a server to support Git pushing and pulling for a given user over SSH, that user needs to be able to execute one or both of those commands via the SSH shell that they are provided on login. On some systems, that shell access is limited to only being able to run those two commands, or even just one of them.

In an ssh:// format URI, it's absolute in the URI, so the / after the host name (or port number) is sent as an argument, which is then read by the remote git-upload-pack exactly as is, so it's effectively an absolute path in the remote filesystem.

```
   git clone ssh://user@example.com/project.git
                |
                v
ssh user@example.com "git-upload-pack '/project.git'"
```

In a "user@host:path" format URI, it's relative to the user's home directory, because the Git client will run:

```
   git clone user@example.com:project.git
                 |
                 v
ssh user@example.com "git-upload-pack 'project.git'"
```

The exception is if a ~ is used, in which case we execute it without the leading /.

```
   ssh://user@example.com/~alice/project.git,
                 |
                 v
ssh user@example.com "git-upload-pack '~alice/project.git'"
```

Depending on the value of the *protocol.version* configuration variable, Git may attempt to send Extra Parameters as a colon-separated string in the GIT_PROTOCOL environment variable. This is done only if the *ssh.variant* configuration variable indicates that the ssh command supports passing environment variables as an argument.

A few things to remember here:

- The "command name" is spelled with dash (e.g. git-upload-pack), but this can be overridden by the client;
- The repository path is always quoted with single quotes.

Fetching Data From a Server

When one Git repository wants to get data that a second repository has, the first can *fetch* from the second. This operation determines what data the server has that the client does not then streams that data down to the client in packfile format.

Reference Discovery

When the client initially connects the server will immediately respond with a version number (if "version=1" is sent as an Extra Parameter), and a listing of each reference it has (all branches and tags) along with the object name that each reference currently points to.

```
$ echo -e -n "0045git-upload-pack /schacon/gitbook.git\0host=example.com
\0\0version=1\0" |
   nc -v example.com 9418
000eversion 1
00887217a7c7e582c46cec22a130adf4b9d7d950fba0 HEAD\0multi_ack thin-pack
```

```
             side-band side-band-64k ofs-delta shallow no-progress include-
tag
00441d3fcd5ced445d1abc402225c0b8a1299641f497 refs/heads/integration
003f7217a7c7e582c46cec22a130adf4b9d7d950fba0 refs/heads/master
003cb88d2441cac0977faf98efc80305012112238d9d refs/tags/v0.9
003c525128480b96c89e6418b1e40909bf6c5b2d580f refs/tags/v1.0
003fe92df48743b7bc7d26bcaabfddde0a1e20cae47c refs/tags/v1.0^{}
0000
```

The returned response is a pkt-line stream describing each ref and its current value. The stream MUST be sorted by name according to the C locale ordering.

If HEAD is a valid ref, HEAD MUST appear as the first advertised ref. If HEAD is not a valid ref, HEAD MUST NOT appear in the advertisement list at all, but other refs may still appear.

The stream MUST include capability declarations behind a NUL on the first ref. The peeled value of a ref (that is "ref^{}") MUST be immediately after the ref itself, if presented. A conforming server MUST peel the ref if it's an annotated tag.

```
advertised-refs  =  *1("version 1")
                    (no-refs / list-of-refs)
                    *shallow
                    flush-pkt

no-refs          =  PKT-LINE(zero-id SP "capabilities^{}"
                    NUL capability-list)

list-of-refs     =  first-ref *other-ref
first-ref        =  PKT-LINE(obj-id SP refname
                    NUL capability-list)

other-ref        =  PKT-LINE(other-tip / other-peeled)
other-tip        =  obj-id SP refname
other-peeled     =  obj-id SP refname "^{}"

shallow          =  PKT-LINE("shallow" SP obj-id)

capability-list  =  capability *(SP capability)
capability       =  1*(LC_ALPHA / DIGIT / "-" / "_")
LC_ALPHA         =  %x61-7A
```

Server and client MUST use lowercase for obj-id, both MUST treat obj-id as case-insensitive.

See protocol-capabilities.txt for a list of allowed server capabilities and descriptions.

Packfile Negotiation

After reference and capabilities discovery, the client can decide to terminate the connection by sending a flush-pkt, telling the server it can now gracefully terminate, and disconnect, when it does not need any pack data. This can happen with the ls-remote command, and also can happen when the client already is up to date.

Otherwise, it enters the negotiation phase, where the client and server determine what the minimal packfile necessary for transport is, by telling the server what objects it wants, its shallow objects (if any), and the maximum commit depth it wants (if any). The client will also send a list of the capabilities it wants to be in effect, out of what the server said it could do with the first *want* line.

```
upload-request    =  want-list
                     *shallow-line
                     *1depth-request
                     [filter-request]
```

```
                      flush-pkt

want-list         =  first-want
                     *additional-want

shallow-line      =  PKT-LINE("shallow" SP obj-id)

depth-request     =  PKT-LINE("deepen" SP depth) /
                     PKT-LINE("deepen-since" SP timestamp) /
                     PKT-LINE("deepen-not" SP ref)

first-want        =  PKT-LINE("want" SP obj-id SP capability-list)
additional-want   =  PKT-LINE("want" SP obj-id)

depth             =  1*DIGIT

filter-request    =  PKT-LINE("filter" SP filter-spec)
```

Clients MUST send all the obj-ids it wants from the reference discovery phase as *want* lines. Clients MUST send at least one *want* command in the request body. Clients MUST NOT mention an obj-id in a *want* command which did not appear in the response obtained through ref discovery.

The client MUST write all obj-ids which it only has shallow copies of (meaning that it does not have the parents of a commit) as *shallow* lines so that the server is aware of the limitations of the client's history.

The client now sends the maximum commit history depth it wants for this transaction, which is the number of commits it wants from the tip of the history, if any, as a *deepen* line. A depth of 0 is the same as not making a depth request. The client does not want to receive any commits beyond this depth, nor does it want objects needed only to complete those commits. Commits whose parents are not received as a result are defined as shallow and marked as such in the server. This information is sent back to the client in the next step.

The client can optionally request that pack-objects omit various objects from the packfile using one of several filtering techniques. These are intended for use with partial clone and partial fetch operations. An object that does not meet a filter-spec value is omitted unless explicitly requested in a *want* line. See *rev-list* for possible filter-spec values.

Once all the *want's and 'shallow's (and optional 'deepen*) are transferred, clients MUST send a flush-pkt, to tell the server side that it is done sending the list.

Otherwise, if the client sent a positive depth request, the server will determine which commits will and will not be shallow and send this information to the client. If the client did not request a positive depth, this step is skipped.

```
shallow-update   =  *shallow-line
                    *unshallow-line
                    flush-pkt

shallow-line     =  PKT-LINE("shallow" SP obj-id)

unshallow-line   =  PKT-LINE("unshallow" SP obj-id)
```

If the client has requested a positive depth, the server will compute the set of commits which are no deeper than the desired depth. The set of commits starts at the client's wants.

The server writes *shallow* lines for each commit whose parents will not be sent as a result. The server writes an *unshallow* line for each commit which the client has indicated is shallow, but is no longer shallow at the currently requested depth (that is, its parents will now be sent). The server MUST NOT mark as unshallow anything which the client has not indicated was shallow.

Now the client will send a list of the obj-ids it has using *have* lines, so the server can make a packfile that only contains the objects that the client needs. In multi_ack mode, the canonical implementation will send up to 32 of

these at a time, then will send a flush-pkt. The canonical implementation will skip ahead and send the next 32 immediately, so that there is always a block of 32 "in-flight on the wire" at a time.

```
upload-haves      =  have-list
                     compute-end

have-list         =  *have-line
have-line         =  PKT-LINE("have" SP obj-id)
compute-end       =  flush-pkt / PKT-LINE("done")
```

If the server reads *have* lines, it then will respond by ACKing any of the obj-ids the client said it had that the server also has. The server will ACK obj-ids differently depending on which ack mode is chosen by the client.

In multi_ack mode:

- the server will respond with *ACK obj-id continue* for any common commits.
- once the server has found an acceptable common base commit and is ready to make a packfile, it will blindly ACK all *have* obj-ids back to the client.
- the server will then send a *NAK* and then wait for another response from the client - either a *done* or another list of *have* lines.

In multi_ack_detailed mode:

- the server will differentiate the ACKs where it is signaling that it is ready to send data with *ACK obj-id ready* lines, and signals the identified common commits with *ACK obj-id common* lines.

Without either multi_ack or multi_ack_detailed:

- upload-pack sends "ACK obj-id" on the first common object it finds. After that it says nothing until the client gives it a "done".
- upload-pack sends "NAK" on a flush-pkt if no common object has been found yet. If one has been found, and thus an ACK was already sent, it's silent on the flush-pkt.

After the client has gotten enough ACK responses that it can determine that the server has enough information to send an efficient packfile (in the canonical implementation, this is determined when it has received enough ACKs that it can color everything left in the --date-order queue as common with the server, or the --date-order queue is empty), or the client determines that it wants to give up (in the canonical implementation, this is determined when the client sends 256 *have* lines without getting any of them ACKed by the server - meaning there is nothing in common and the server should just send all of its objects), then the client will send a *done* command. The *done* command signals to the server that the client is ready to receive its packfile data.

However, the 256 limit **only** turns on in the canonical client implementation if we have received at least one "ACK %s continue" during a prior round. This helps to ensure that at least one common ancestor is found before we give up entirely.

Once the *done* line is read from the client, the server will either send a final *ACK obj-id* or it will send a *NAK*. *obj-id* is the object name of the last commit determined to be common. The server only sends ACK after *done* if there is at least one common base and multi_ack or multi_ack_detailed is enabled. The server always sends NAK after *done* if there is no common base found.

Instead of *ACK* or *NAK*, the server may send an error message (for example, if it does not recognize an object in a *want* line received from the client).

Then the server will start sending its packfile data.

```
server-response = *ack_multi ack / nak
ack_multi       = PKT-LINE("ACK" SP obj-id ack_status)
```

```
ack_status      = "continue" / "common" / "ready"
ack             = PKT-LINE("ACK" SP obj-id)
nak             = PKT-LINE("NAK")
```

A simple clone may look like this (with no *have* lines):

```
C: 0054want 74730d410fcb6603ace96f1dc55ea6196122532d multi_ack \
  side-band-64k ofs-delta\n
C: 0032want 7d1665144a3a975c05f1f43902ddaf084e784dbe\n
C: 0032want 5a3f6be755bbb7deae50065988cbfa1ffa9ab68a\n
C: 0032want 7e47fe2bd8d01d481f44d7af0531bd93d3b21c01\n
C: 0032want 74730d410fcb6603ace96f1dc55ea6196122532d\n
C: 0000
C: 0009done\n

S: 0008NAK\n
S: [PACKFILE]
```

An incremental update (fetch) response might look like this:

```
C: 0054want 74730d410fcb6603ace96f1dc55ea6196122532d multi_ack \
  side-band-64k ofs-delta\n
C: 0032want 7d1665144a3a975c05f1f43902ddaf084e784dbe\n
C: 0032want 5a3f6be755bbb7deae50065988cbfa1ffa9ab68a\n
C: 0000
C: 0032have 7e47fe2bd8d01d481f44d7af0531bd93d3b21c01\n
C: [30 more have lines]
C: 0032have 74730d410fcb6603ace96f1dc55ea6196122532d\n
C: 0000

S: 003aACK 7e47fe2bd8d01d481f44d7af0531bd93d3b21c01 continue\n
S: 003aACK 74730d410fcb6603ace96f1dc55ea6196122532d continue\n
S: 0008NAK\n

C: 0009done\n

S: 0031ACK 74730d410fcb6603ace96f1dc55ea6196122532d\n
S: [PACKFILE]
```

Packfile Data

Now that the client and server have finished negotiation about what the minimal amount of data that needs to be sent to the client is, the server will construct and send the required data in packfile format.

See Section G.5.5, “gitformat-pack(5)” for what the packfile itself actually looks like.

If *side-band* or *side-band-64k* capabilities have been specified by the client, the server will send the packfile data multiplexed.

Each packet starting with the packet-line length of the amount of data that follows, followed by a single byte specifying the sideband the following data is coming in on.

In *side-band* mode, it will send up to 999 data bytes plus 1 control code, for a total of up to 1000 bytes in a pkt-line. In *side-band-64k* mode it will send up to 65519 data bytes plus 1 control code, for a total of up to 65520 bytes in a pkt-line.

The sideband byte will be a *1*, *2* or a *3*. Sideband *1* will contain packfile data, sideband *2* will be used for progress information that the client will generally print to stderr and sideband *3* is used for error information.

If no *side-band* capability was specified, the server will stream the entire packfile without multiplexing.

Pushing Data To a Server

Pushing data to a server will invoke the *receive-pack* process on the server, which will allow the client to tell it which references it should update and then send all the data the server will need for those new references to be complete. Once all the data is received and validated, the server will then update its references to what the client specified.

Authentication

The protocol itself contains no authentication mechanisms. That is to be handled by the transport, such as SSH, before the *receive-pack* process is invoked. If *receive-pack* is configured over the Git transport, those repositories will be writable by anyone who can access that port (9418) as that transport is unauthenticated.

Reference Discovery

The reference discovery phase is done nearly the same way as it is in the fetching protocol. Each reference obj-id and name on the server is sent in packet-line format to the client, followed by a flush-pkt. The only real difference is that the capability listing is different - the only possible values are *report-status*, *report-status-v2*, *delete-refs*, *ofs-delta*, *atomic* and *push-options*.

Reference Update Request and Packfile Transfer

Once the client knows what references the server is at, it can send a list of reference update requests. For each reference on the server that it wants to update, it sends a line listing the obj-id currently on the server, the obj-id the client would like to update it to and the name of the reference.

This list is followed by a flush-pkt.

```
update-requests   =  *shallow ( command-list | push-cert )

shallow           =  PKT-LINE("shallow" SP obj-id)

command-list      =  PKT-LINE(command NUL capability-list)
                     *PKT-LINE(command)
                     flush-pkt

command           =  create / delete / update
create            =  zero-id SP new-id  SP name
delete            =  old-id  SP zero-id SP name
update            =  old-id  SP new-id  SP name

old-id            =  obj-id
new-id            =  obj-id

push-cert         = PKT-LINE("push-cert" NUL capability-list LF)
                    PKT-LINE("certificate version 0.1" LF)
                    PKT-LINE("pusher" SP ident LF)
                    PKT-LINE("pushee" SP url LF)
                    PKT-LINE("nonce" SP nonce LF)
                    *PKT-LINE("push-option" SP push-option LF)
                    PKT-LINE(LF)
                    *PKT-LINE(command LF)
                    *PKT-LINE(gpg-signature-lines LF)
                    PKT-LINE("push-cert-end" LF)

push-option       =  1*( VCHAR | SP )
```

If the server has advertised the *push-options* capability and the client has specified *push-options* as part of the capability list above, the client then sends its push options followed by a flush-pkt.

```
push-options       =  *PKT-LINE(push-option) flush-pkt
```

For backwards compatibility with older Git servers, if the client sends a push cert and push options, it MUST send its push options both embedded within the push cert and after the push cert. (Note that the push options within the cert are prefixed, but the push options after the cert are not.) Both these lists MUST be the same, modulo the prefix.

After that the packfile that should contain all the objects that the server will need to complete the new references will be sent.

```
packfile           =  "PACK" 28*(OCTET)
```

If the receiving end does not support delete-refs, the sending end MUST NOT ask for delete command.

If the receiving end does not support push-cert, the sending end MUST NOT send a push-cert command. When a push-cert command is sent, command-list MUST NOT be sent; the commands recorded in the push certificate is used instead.

The packfile MUST NOT be sent if the only command used is *delete*.

A packfile MUST be sent if either create or update command is used, even if the server already has all the necessary objects. In this case the client MUST send an empty packfile. The only time this is likely to happen is if the client is creating a new branch or a tag that points to an existing obj-id.

The server will receive the packfile, unpack it, then validate each reference that is being updated that it hasn't changed while the request was being processed (the obj-id is still the same as the old-id), and it will run any update hooks to make sure that the update is acceptable. If all of that is fine, the server will then update the references.

Push Certificate

A push certificate begins with a set of header lines. After the header and an empty line, the protocol commands follow, one per line. Note that the trailing LF in push-cert PKT-LINEs is *not* optional; it must be present.

Currently, the following header fields are defined:

pusher ident

Identify the GPG key in "Human Readable Name <email@address [mailto:email@address]>" format.

pushee url

The repository URL (anonymized, if the URL contains authentication material) the user who ran *git push* intended to push into.

nonce nonce

The *nonce* string the receiving repository asked the pushing user to include in the certificate, to prevent replay attacks.

The GPG signature lines are a detached signature for the contents recorded in the push certificate before the signature block begins. The detached signature is used to certify that the commands were given by the pusher, who must be the signer.

Report Status

After receiving the pack data from the sender, the receiver sends a report if *report-status* or *report-status-v2* capability is in effect. It is a short listing of what happened in that update. It will first list the status of the packfile unpacking as either *unpack ok* or *unpack [error]*. Then it will list the status for each of the references that it tried to update. Each line is either *ok [refname]* if the update was successful, or *ng [refname] [error]* if the update was not.

```
report-status      = unpack-status
                     1*(command-status)
                     flush-pkt
```

```
  unpack-status     = PKT-LINE("unpack" SP unpack-result)
  unpack-result     = "ok" / error-msg

  command-status    = command-ok / command-fail
  command-ok        = PKT-LINE("ok" SP refname)
  command-fail      = PKT-LINE("ng" SP refname SP error-msg)

  error-msg         = 1*(OCTET) ; where not "ok"
```

The *report-status-v2* capability extends the protocol by adding new option lines in order to support reporting of reference rewritten by the *proc-receive* hook. The *proc-receive* hook may handle a command for a pseudo-reference which may create or update one or more references, and each reference may have different name, different new-oid, and different old-oid.

```
  report-status-v2  = unpack-status
                      1*(command-status-v2)
                      flush-pkt

  unpack-status     = PKT-LINE("unpack" SP unpack-result)
  unpack-result     = "ok" / error-msg

  command-status-v2 = command-ok-v2 / command-fail
  command-ok-v2     = command-ok
                      *option-line

  command-ok        = PKT-LINE("ok" SP refname)
  command-fail      = PKT-LINE("ng" SP refname SP error-msg)

  error-msg         = 1*(OCTET) ; where not "ok"

  option-line       = *1(option-refname)
                      *1(option-old-oid)
                      *1(option-new-oid)
                      *1(option-forced-update)

  option-refname    = PKT-LINE("option" SP "refname" SP refname)
  option-old-oid    = PKT-LINE("option" SP "old-oid" SP obj-id)
  option-new-oid    = PKT-LINE("option" SP "new-oid" SP obj-id)
  option-force      = PKT-LINE("option" SP "forced-update")
```

Updates can be unsuccessful for a number of reasons. The reference can have changed since the reference discovery phase was originally sent, meaning someone pushed in the meantime. The reference being pushed could be a non-fast-forward reference and the update hooks or configuration could be set to not allow that, etc. Also, some references can be updated while others can be rejected.

An example client/server communication might look like this:

```
   S: 006274730d410fcb6603ace96f1dc55ea6196122532d refs/heads/local
\0report-status delete-refs ofs-delta\n
   S: 003e7d1665144a3a975c05f1f43902ddaf084e784dbe refs/heads/debug\n
   S: 003f74730d410fcb6603ace96f1dc55ea6196122532d refs/heads/master\n
   S: 003d74730d410fcb6603ace96f1dc55ea6196122532d refs/heads/team\n
   S: 0000

   C: 00677d1665144a3a975c05f1f43902ddaf084e784dbe
 74730d410fcb6603ace96f1dc55ea6196122532d refs/heads/debug\n
   C: 006874730d410fcb6603ace96f1dc55ea6196122532d
 5a3f6be755bbb7deae50065988cbfa1ffa9ab68a refs/heads/master\n
```

```
C: 0000
C: [PACKDATA]

S: 000eunpack ok\n
S: 0018ok refs/heads/debug\n
S: 002ang refs/heads/master non-fast-forward\n
```

GIT

Part of the Section G.3.1, “git(1)” suite

G.5.11. gitprotocol-v2(5)

2024-07-02

Revision History

2024-07-02

NAME

gitprotocol-v2 - Git Wire Protocol, Version 2

SYNOPSIS

<over-the-wire-protocol>

DESCRIPTION

This document presents a specification for a version 2 of Git's wire protocol. Protocol v2 will improve upon v1 in the following ways:

- Instead of multiple service names, multiple commands will be supported by a single service
- Easily extendable as capabilities are moved into their own section of the protocol, no longer being hidden behind a NUL byte and limited by the size of a pkt-line
- Separate out other information hidden behind NUL bytes (e.g. agent string as a capability and symrefs can be requested using *ls-refs*)
- Reference advertisement will be omitted unless explicitly requested
- ls-refs command to explicitly request some refs
- Designed with http and stateless-rpc in mind. With clear flush semantics the http remote helper can simply act as a proxy

In protocol v2 communication is command oriented. When first contacting a server a list of capabilities will be advertised. Some of these capabilities will be commands which a client can request be executed. Once a command has completed, a client can reuse the connection and request that other commands be executed.

Packet-Line Framing

All communication is done using packet-line framing, just as in v1. See Section G.5.10, “gitprotocol-pack(5)” and Section G.5.8, “gitprotocol-common(5)” for more information.

In protocol v2 these special packets will have the following semantics:

- *0000* Flush Packet (flush-pkt) - indicates the end of a message
- *0001* Delimiter Packet (delim-pkt) - separates sections of a message
- *0002* Response End Packet (response-end-pkt) - indicates the end of a response for stateless connections

Initial Client Request

In general a client can request to speak protocol v2 by sending *version=2* through the respective side-channel for the transport being used which inevitably sets *GIT_PROTOCOL*. More information can be found in Section G.5.10, “gitprotocol-pack(5)” and Section G.5.9, “gitprotocol-http(5)”, as well as the *GIT_PROTOCOL* definition in *git.txt*. In all cases the response from the server is the capability advertisement.

1. Git Transport

When using the git:// transport, you can request to use protocol v2 by sending "version=2" as an extra parameter:

```
003egit-upload-pack /project.git\0host=myserver.com\0\0version=2\0
```

2. SSH and File Transport

When using either the ssh:// or file:// transport, the GIT_PROTOCOL environment variable must be set explicitly to include "version=2". The server may need to be configured to allow this environment variable to pass.

3. HTTP Transport

When using the http:// or https:// transport a client makes a "smart" info/refs request as described in Section G.5.9, “gitprotocol-http(5)” and requests that v2 be used by supplying "version=2" in the *Git-Protocol* header.

```
C: GET $GIT_URL/info/refs?service=git-upload-pack HTTP/1.0
C: Git-Protocol: version=2
```

A v2 server would reply:

```
S: 200 OK
S: <Some headers>
S: ...
S:
S: 000eversion 2\n
S: <capability-advertisement>
```

Subsequent requests are then made directly to the service *$GIT_URL/git-upload-pack*. (This works the same for git-receive-pack).

Uses the *--http-backend-info-refs* option to Section G.3.151, “git-upload-pack(1)”.

The server may need to be configured to pass this header's contents via the *GIT_PROTOCOL* variable. See the discussion in *git-http-backend.txt*.

Capability Advertisement

A server which decides to communicate (based on a request from a client) using protocol version 2, notifies the client by sending a version string in its initial response followed by an advertisement of its capabilities. Each capability is a key with an optional value. Clients must ignore all unknown keys. Semantics of unknown values are left to the definition of each key. Some capabilities will describe commands which can be requested to be executed by the client.

```
capability-advertisement = protocol-version
                           capability-list
                           flush-pkt

protocol-version = PKT-LINE("version 2" LF)
capability-list = *capability
capability = PKT-LINE(key[=value] LF)

key = 1*(ALPHA | DIGIT | "-_")
value = 1*(ALPHA | DIGIT | " -_.,?\/{}[]()<>!@#$%^&*+=:;")
```

Command Request

After receiving the capability advertisement, a client can then issue a request to select the command it wants with any particular capabilities or arguments. There is then an optional section where the client can provide any command specific parameters or queries. Only a single command can be requested at a time.

```
request = empty-request | command-request
empty-request = flush-pkt
command-request = command
                  capability-list
                  delim-pkt
                  command-args
                  flush-pkt
command = PKT-LINE("command=" key LF)
command-args = *command-specific-arg

command-specific-args are packet line framed arguments defined by
each individual command.
```

The server will then check to ensure that the client's request is comprised of a valid command as well as valid capabilities which were advertised. If the request is valid the server will then execute the command. A server MUST wait till it has received the client's entire request before issuing a response. The format of the response is determined by the command being executed, but in all cases a flush-pkt indicates the end of the response.

When a command has finished, and the client has received the entire response from the server, a client can either request that another command be executed or can terminate the connection. A client may optionally send an empty request consisting of just a flush-pkt to indicate that no more requests will be made.

Capabilities

There are two different types of capabilities: normal capabilities, which can be used to convey information or alter the behavior of a request, and commands, which are the core actions that a client wants to perform (fetch, push, etc).

Protocol version 2 is stateless by default. This means that all commands must only last a single round and be stateless from the perspective of the server side, unless the client has requested a capability indicating that state should be maintained by the server. Clients MUST NOT require state management on the server side in order to function correctly. This permits simple round-robin load-balancing on the server side, without needing to worry about state management.

1. agent

The server can advertise the *agent* capability with a value *X* (in the form *agent=X*) to notify the client that the server is running version *X*. The client may optionally send its own agent string by including the *agent* capability with a value *Y* (in the form *agent=Y*) in its request to the server (but it MUST NOT do so if the server did not advertise the agent capability). The *X* and *Y* strings may contain any printable ASCII characters except space (i.e., the byte range 32 < x < 127), and are typically of the form "package/version" (e.g., "git/1.8.3.1"). The agent strings are purely informative for statistics and debugging purposes, and MUST NOT be used to programmatically assume the presence or absence of particular features.

2. ls-refs

ls-refs is the command used to request a reference advertisement in v2. Unlike the current reference advertisement, ls-refs takes in arguments which can be used to limit the refs sent from the server.

Additional features not supported in the base command will be advertised as the value of the command in the capability advertisement in the form of a space separated list of features: "<command>=<feature-1> <feature-2>"

ls-refs takes in the following arguments:

```
symrefs
```

```
    In addition to the object pointed by it, show the underlying ref
    pointed by it when showing a symbolic ref.
peel
    Show peeled tags.
ref-prefix <prefix>
    When specified, only references having a prefix matching one of
    the provided prefixes are displayed. Multiple instances may be
    given, in which case references matching any prefix will be
    shown. Note that this is purely for optimization; a server MAY
    show refs not matching the prefix if it chooses, and clients
    should filter the result themselves.
```

If the *unborn* feature is advertised the following argument can be included in the client's request.

```
unborn
    The server will send information about HEAD even if it is a symref
    pointing to an unborn branch in the form "unborn HEAD
    symref-target:<target>".
```

The output of ls-refs is as follows:

```
output = *ref
         flush-pkt
obj-id-or-unborn = (obj-id | "unborn")
ref = PKT-LINE(obj-id-or-unborn SP refname *(SP ref-attribute) LF)
ref-attribute = (symref | peeled)
symref = "symref-target:" symref-target
peeled = "peeled:" obj-id
```

3. fetch

fetch is the command used to fetch a packfile in v2. It can be looked at as a modified version of the v1 fetch where the ref-advertisement is stripped out (since the *ls-refs* command fills that role) and the message format is tweaked to eliminate redundancies and permit easy addition of future extensions.

Additional features not supported in the base command will be advertised as the value of the command in the capability advertisement in the form of a space separated list of features: "<command>=<feature-1> <feature-2>"

A *fetch* request can take the following arguments:

```
want <oid>
    Indicates to the server an object which the client wants to
    retrieve.  Wants can be anything and are not limited to
    advertised objects.

have <oid>
    Indicates to the server an object which the client has locally.
    This allows the server to make a packfile which only contains
    the objects that the client needs. Multiple 'have' lines can be
    supplied.

done
    Indicates to the server that negotiation should terminate (or
    not even begin if performing a clone) and that the server should
    use the information supplied in the request to construct the
    packfile.

thin-pack
    Request that a thin pack be sent, which is a pack with deltas
    which reference base objects not contained within the pack (but
```

```
    are known to exist at the receiving end). This can reduce the
    network traffic significantly, but it requires the receiving end
    to know how to "thicken" these packs by adding the missing bases
    to the pack.

no-progress
    Request that progress information that would normally be sent on
    side-band channel 2, during the packfile transfer, should not be
    sent.  However, the side-band channel 3 is still used for error
    responses.

include-tag
    Request that annotated tags should be sent if the objects they
    point to are being sent.

ofs-delta
    Indicate that the client understands PACKv2 with delta referring
    to its base by position in pack rather than by an oid.  That is,
    they can read OBJ_OFS_DELTA (aka type 6) in a packfile.
```

If the *shallow* feature is advertised the following arguments can be included in the clients request as well as the potential addition of the *shallow-info* section in the server's response as explained below.

```
shallow <oid>
    A client must notify the server of all commits for which it only
    has shallow copies (meaning that it doesn't have the parents of
    a commit) by supplying a 'shallow <oid>' line for each such
    object so that the server is aware of the limitations of the
    client's history.  This is so that the server is aware that the
    client may not have all objects reachable from such commits.

deepen <depth>
    Requests that the fetch/clone should be shallow having a commit
    depth of <depth> relative to the remote side.

deepen-relative
    Requests that the semantics of the "deepen" command be changed
    to indicate that the depth requested is relative to the client's
    current shallow boundary, instead of relative to the requested
    commits.

deepen-since <timestamp>
    Requests that the shallow clone/fetch should be cut at a
    specific time, instead of depth.  Internally it's equivalent to
    doing "git rev-list --max-age=<timestamp>". Cannot be used with
    "deepen".

deepen-not <rev>
    Requests that the shallow clone/fetch should be cut at a
    specific revision specified by '<rev>', instead of a depth.
    Internally it's equivalent of doing "git rev-list --not <rev>".
    Cannot be used with "deepen", but can be used with
    "deepen-since".
```

If the *filter* feature is advertised, the following argument can be included in the client's request:

```
filter <filter-spec>
    Request that various objects from the packfile be omitted
    using one of several filtering techniques. These are intended
    for use with partial clone and partial fetch operations. See
```

```
    `rev-list` for possible "filter-spec" values. When communicating
    with other processes, senders SHOULD translate scaled integers
    (e.g. "1k") into a fully-expanded form (e.g. "1024") to aid
    interoperability with older receivers that may not understand
    newly-invented scaling suffixes. However, receivers SHOULD
    accept the following suffixes: 'k', 'm', and 'g' for 1024,
    1048576, and 1073741824, respectively.
```

If the *ref-in-want* feature is advertised, the following argument can be included in the client's request as well as the potential addition of the *wanted-refs* section in the server's response as explained below.

```
want-ref <ref>
    Indicates to the server that the client wants to retrieve a
    particular ref, where <ref> is the full name of a ref on the
    server.  It is a protocol error to send want-ref for the
    same ref more than once.
```

If the *sideband-all* feature is advertised, the following argument can be included in the client's request:

```
sideband-all
    Instruct the server to send the whole response multiplexed, not just
    the packfile section. All non-flush and non-delim PKT-LINE in the
    response (not only in the packfile section) will then start with a byte
    indicating its sideband (1, 2, or 3), and the server may send "0005\2"
    (a PKT-LINE of sideband 2 with no payload) as a keepalive packet.
```

If the *packfile-uris* feature is advertised, the following argument can be included in the client's request as well as the potential addition of the *packfile-uris* section in the server's response as explained below. Note that at most one *packfile-uris* line can be sent to the server.

```
packfile-uris <comma-separated-list-of-protocols>
    Indicates to the server that the client is willing to receive
    URIs of any of the given protocols in place of objects in the
    sent packfile. Before performing the connectivity check, the
    client should download from all given URIs. Currently, the
    protocols supported are "http" and "https".
```

If the *wait-for-done* feature is advertised, the following argument can be included in the client's request.

```
wait-for-done
    Indicates to the server that it should never send "ready", but
    should wait for the client to say "done" before sending the
    packfile.
```

The response of *fetch* is broken into a number of sections separated by delimiter packets (0001), with each section beginning with its section header. Most sections are sent only when the packfile is sent.

```
output = acknowledgements flush-pkt |
         [acknowledgments delim-pkt] [shallow-info delim-pkt]
         [wanted-refs delim-pkt] [packfile-uris delim-pkt]
         packfile flush-pkt

acknowledgments = PKT-LINE("acknowledgments" LF)
                  (nak | *ack)
                  (ready)
ready = PKT-LINE("ready" LF)
nak = PKT-LINE("NAK" LF)
ack = PKT-LINE("ACK" SP obj-id LF)

shallow-info = PKT-LINE("shallow-info" LF)
```

```
               *PKT-LINE((shallow | unshallow) LF)
shallow = "shallow" SP obj-id
unshallow = "unshallow" SP obj-id

wanted-refs = PKT-LINE("wanted-refs" LF)
              *PKT-LINE(wanted-ref LF)
wanted-ref = obj-id SP refname

packfile-uris = PKT-LINE("packfile-uris" LF) *packfile-uri
packfile-uri = PKT-LINE(40*(HEXDIGIT) SP *%x20-ff LF)

packfile = PKT-LINE("packfile" LF)
           *PKT-LINE(%x01-03 *%x00-ff)

acknowledgments section
    * If the client determines that it is finished with negotiations by
      sending a "done" line (thus requiring the server to send a packfile),
      the acknowledgments sections MUST be omitted from the server's
      response.
```

- Always begins with the section header "acknowledgments"
- The server will respond with "NAK" if none of the object ids sent as have lines were common.
- The server will respond with "ACK obj-id" for all of the object ids sent as have lines which are common.
- A response cannot have both "ACK" lines as well as a "NAK" line.
- The server will respond with a "ready" line indicating that the server has found an acceptable common base and is ready to make and send a packfile (which will be found in the packfile section of the same response)
- If the server has found a suitable cut point and has decided to send a "ready" line, then the server can decide to (as an optimization) omit any "ACK" lines it would have sent during its response. This is because the server will have already determined the objects it plans to send to the client and no further negotiation is needed.

```
shallow-info section
    * If the client has requested a shallow fetch/clone, a shallow
      client requests a fetch or the server is shallow then the
      server's response may include a shallow-info section.  The
      shallow-info section will be included if (due to one of the
      above conditions) the server needs to inform the client of any
      shallow boundaries or adjustments to the clients already
      existing shallow boundaries.
```

- Always begins with the section header "shallow-info"
- If a positive depth is requested, the server will compute the set of commits which are no deeper than the desired depth.
- The server sends a "shallow obj-id" line for each commit whose parents will not be sent in the following packfile.
- The server sends an "unshallow obj-id" line for each commit which the client has indicated is shallow, but is no longer shallow as a result of the fetch (due to its parents being sent in the following packfile).
- The server MUST NOT send any "unshallow" lines for anything which the client has not indicated was shallow as a part of its request.

```
wanted-refs section
    * This section is only included if the client has requested a
      ref using a 'want-ref' line and if a packfile section is also
      included in the response.
```

- Always begins with the section header "wanted-refs".
- The server will send a ref listing ("<oid> <refname>") for each reference requested using *want-ref* lines.
- The server MUST NOT send any refs which were not requested using *want-ref* lines.

```
packfile-uris section
    * This section is only included if the client sent
      'packfile-uris' and the server has at least one such URI to
      send.
```

- Always begins with the section header "packfile-uris".
- For each URI the server sends, it sends a hash of the pack's contents (as output by git index-pack) followed by the URI.
- The hashes are 40 hex characters long. When Git upgrades to a new hash algorithm, this might need to be updated. (It should match whatever index-pack outputs after "pack\t" or "keep\t".

```
packfile section
    * This section is only included if the client has sent 'want'
      lines in its request and either requested that no more
      negotiation be done by sending 'done' or if the server has
      decided it has found a sufficient cut point to produce a
      packfile.
```

- Always begins with the section header "packfile"
- The transmission of the packfile begins immediately after the section header
- The data transfer of the packfile is always multiplexed, using the same semantics of the *side-band-64k* capability from protocol version 1. This means that each packet, during the packfile data stream, is made up of a leading 4-byte pkt-line length (typical of the pkt-line format), followed by a 1-byte stream code, followed by the actual data.

```
The stream code can be one of:
      1 - pack data
      2 - progress messages
      3 - fatal error message just before stream aborts
```

4. server-option

If advertised, indicates that any number of server specific options can be included in a request. This is done by sending each option as a "server-option=<option>" capability line in the capability-list section of a request.

The provided options must not contain a NUL or LF character.

5. object-format

The server can advertise the *object-format* capability with a value *X* (in the form *object-format=X*) to notify the client that the server is able to deal with objects using hash algorithm X. If not specified, the server is assumed to only handle SHA-1. If the client would like to use a hash algorithm other than SHA-1, it should specify its object-format string.

6. session-id=<session-id>

The server may advertise a session ID that can be used to identify this process across multiple requests. The client may advertise its own session ID back to the server as well.

Session IDs should be unique to a given process. They must fit within a packet-line, and must not contain non-printable or whitespace characters. The current implementation uses trace2 session IDs (see *api-trace2* [https://

www.kernel.org/pub/software/scm/git/docs/technical/api-trace2.html] for details), but this may change and users of the session ID should not rely on this fact.

7. object-info

object-info is the command to retrieve information about one or more objects. Its main purpose is to allow a client to make decisions based on this information without having to fully fetch objects. Object size is the only information that is currently supported.

An *object-info* request takes the following arguments:

```
size
Requests size information to be returned for each listed object id.

oid <oid>
Indicates to the server an object which the client wants to obtain
information for.
```

The response of *object-info* is a list of the requested object ids and associated requested information, each separated by a single space.

```
output = info flush-pkt

info = PKT-LINE(attrs) LF)
        *PKT-LINE(obj-info LF)

attrs = attr | attrs SP attrs

attr = "size"

obj-info = obj-id SP obj-size
```

8. bundle-uri

If the *bundle-uri* capability is advertised, the server supports the bundle-uri command.

The capability is currently advertised with no value (i.e. not "bundle-uri=somevalue"), a value may be added in the future for supporting command-wide extensions. Clients MUST ignore any unknown capability values and proceed with the 'bundle-uri` dialog they support.

The *bundle-uri* command is intended to be issued before *fetch* to get URIs to bundle files (see Section G.3.12, “git-bundle(1)”) to "seed" and inform the subsequent *fetch* command.

The client CAN issue *bundle-uri* before or after any other valid command. To be useful to clients it's expected that it'll be issued after an *ls-refs* and before *fetch*, but CAN be issued at any time in the dialog.

8.1. DISCUSSION of bundle-uri

The intent of the feature is optimize for server resource consumption in the common case by changing the common case of fetching a very large PACK during Section G.3.24, “git-clone(1)” into a smaller incremental fetch.

It also allows servers to achieve better caching in combination with an *uploadpack.packObjectsHook* (see Section G.3.29, “git-config(1)”).

By having new clones or fetches be a more predictable and common negotiation against the tips of recently produces *.bundle file(s). Servers might even pre-generate the results of such negotiations for the *uploadpack.packObjectsHook* as new pushes come in.

One way that servers could take advantage of these bundles is that the server would anticipate that fresh clones will download a known bundle, followed by catching up to the current state of the repository using ref tips found in that bundle (or bundles).

8.2. PROTOCOL for bundle-uri

A *bundle-uri* request takes no arguments, and as noted above does not currently advertise a capability value. Both may be added in the future.

When the client issues a *command=bundle-uri* request, the response is a list of key-value pairs provided as packet lines with value *<key>=<value>*. Each *<key>* should be interpreted as a config key from the *bundle.** namespace to construct a list of bundles. These keys are grouped by a *bundle.<id>.* subsection, where each key corresponding to a given *<id>* contributes attributes to the bundle defined by that *<id>*. See Section G.3.29, “git-config(1)” for the specific details of these keys and how the Git client will interpret their values.

Clients MUST parse the line according to the above format, lines that do not conform to the format SHOULD be discarded. The user MAY be warned in such a case.

8.3. bundle-uri CLIENT AND SERVER EXPECTATIONS

URI CONTENTS

The content at the advertised URIs MUST be one of two types.

The advertised URI may contain a bundle file that *git bundle verify* would accept. I.e. they MUST contain one or more reference tips for use by the client, MUST indicate prerequisites (in any) with standard "-" prefixes, and MUST indicate their "object-format", if applicable.

The advertised URI may alternatively contain a plaintext file that *git config --list* would accept (with the *--file* option). The key-value pairs in this list are in the *bundle.** namespace (see Section G.3.29, “git-config(1)”).

bundle-uri CLIENT ERROR RECOVERY

A client MUST above all gracefully degrade on errors, whether that error is because of bad missing/data in the bundle URI(s), because that client is too dumb to e.g. understand and fully parse out bundle headers and their prerequisite relationships, or something else.

Server operators should feel confident in turning on "bundle-uri" and not worry if e.g. their CDN goes down that clones or fetches will run into hard failures. Even if the server bundle(s) are incomplete, or bad in some way the client should still end up with a functioning repository, just as if it had chosen not to use this protocol extension.

All subsequent discussion on client and server interaction MUST keep this in mind.

bundle-uri SERVER TO CLIENT

The ordering of the returned bundle uris is not significant. Clients MUST parse their headers to discover their contained OIDS and prerequisites. A client MUST consider the content of the bundle(s) themselves and their header as the ultimate source of truth.

A server MAY even return bundle(s) that don't have any direct relationship to the repository being cloned (either through accident, or intentional "clever" configuration), and expect a client to sort out what data they'd like from the bundle(s), if any.

bundle-uri CLIENT TO SERVER

The client SHOULD provide reference tips found in the bundle header(s) as *have* lines in any subsequent *fetch* request. A client MAY also ignore the bundle(s) entirely if doing so is deemed worse for some reason, e.g. if the bundles can't be downloaded, it doesn't like the tips it finds etc.

WHEN ADVERTISED BUNDLE(S) REQUIRE NO FURTHER NEGOTIATION

If after issuing *bundle-uri* and *ls-refs*, and getting the header(s) of the bundle(s) the client finds that the ref tips it wants can be retrieved entirely from advertised bundle(s), the client MAY disconnect from the Git server. The results of such a *clone* or *fetch* should be indistinguishable from the state attained without using bundle-uri.

EARLY CLIENT DISCONNECTIONS AND ERROR RECOVERY

A client MAY perform an early disconnect while still downloading the bundle(s) (having streamed and parsed their headers). In such a case the client MUST gracefully recover from any errors related to finishing the download and validation of the bundle(s).

I.e. a client might need to re-connect and issue a *fetch* command, and possibly fall back to not making use of *bundle-uri* at all.

This "MAY" behavior is specified as such (and not a "SHOULD") on the assumption that a server advertising bundle uris is more likely than not to be serving up a relatively large repository, and to be pointing to URIs that have a good chance of being in working order. A client MAY e.g. look at the payload size of the bundles as a heuristic to see if an early disconnect is worth it, should falling back on a full "fetch" dialog be necessary.

WHEN ADVERTISED BUNDLE(S) REQUIRE FURTHER NEGOTIATION

A client SHOULD commence a negotiation of a PACK from the server via the "fetch" command using the OID tips found in advertised bundles, even if's still in the process of downloading those bundle(s).

This allows for aggressive early disconnects from any interactive server dialog. The client blindly trusts that the advertised OID tips are relevant, and issues them as *have* lines, it then requests any tips it would like (usually from the "ls-refs" advertisement) via *want* lines. The server will then compute a (hopefully small) PACK with the expected difference between the tips from the bundle(s) and the data requested.

The only connection the client then needs to keep active is to the concurrently downloading static bundle(s), when those and the incremental PACK are retrieved they should be inflated and validated. Any errors at this point should be gracefully recovered from, see above.

8.4. bundle-uri PROTOCOL FEATURES

The client constructs a bundle list from the *<key>=<value>* pairs provided by the server. These pairs are part of the *bundle.** namespace as documented in Section G.3.29, “git-config(1)”. In this section, we discuss some of these keys and describe the actions the client will do in response to this information.

In particular, the *bundle.version* key specifies an integer value. The only accepted value at the moment is *1*, but if the client sees an unexpected value here then the client MUST ignore the bundle list.

As long as *bundle.version* is understood, all other unknown keys MAY be ignored by the client. The server will guarantee compatibility with older clients, though newer clients may be better able to use the extra keys to minimize downloads.

Any backwards-incompatible addition of pre-URI key-value will be guarded by a new *bundle.version* value or values in *bundle-uri* capability advertisement itself, and/or by new future *bundle-uri* request arguments.

Some example key-value pairs that are not currently implemented but could be implemented in the future include:

- Add a "hash=<val>" or "size=<bytes>" advertise the expected hash or size of the bundle file.
- Advertise that one or more bundle files are the same (to e.g. have clients round-robin or otherwise choose one of N possible files).
- A "oid=<OID>" shortcut and "prerequisite=<OID>" shortcut. For expressing the common case of a bundle with one tip and no prerequisites, or one tip and one prerequisite.

  This would allow for optimizing the common case of servers who'd like to provide one "big bundle" containing only their "main" branch, and/or incremental updates thereof.

  A client receiving such a a response MAY assume that they can skip retrieving the header from a bundle at the indicated URI, and thus save themselves and the server(s) the request(s) needed to inspect the headers of that bundle or bundles.

GIT

Part of the Section G.3.1, “git(1)” suite

Glossary

Add	A Git command that is used to add a file to your working tree. The new items are added to the repository when you commit.
BASE revision	This is the common ancestor's version of a conflicted file.
Blame	This command is for text files only, and it annotates every line to show the repository revision in which it was last changed, and the author who made that change. Our GUI implementation is called TortoiseGitBlame and it also shows the commit date/time and the log message when you hover the mouse of the revision number.
Branch	A term frequently used in revision control systems to describe what happens when development forks at a particular point and follows 2 separate paths. You can create a branch off the main development line so as to develop a new feature without rendering the main line unstable. Or you can branch a stable release to which you make only bug fixes, while new developments take place on the unstable trunk. In Git a branch is implemented as a “pointer to a revision”.
Cleanup	Remove untracked files from the working tree. *This is different to TortoiseSVN cleanup*
Clone	A Git command which creates a local working tree in an empty directory by downloading a remote repository.
Commit	This Git command is used to pass the changes in your local working tree back into the repository, creating a new repository revision.
Conflict	When changes from the repository are merged with local changes, sometimes those changes occur on the same lines. In this case Git cannot automatically decide which version to use and the file is said to be in conflict. You have to edit the file manually and resolve the conflict before you can commit any further changes.
Copy	In a Git repository you can manually create a copy of a single file or an entire tree w/o problems.
Delete	When you delete a versioned item (and commit the change) the item no longer exists in the repository after the committed revision. But of course it still exists in earlier repository revisions, so you can still access it. If necessary, you can copy a deleted item and “resurrect” it complete with history.
Diff	Shorthand for “Show Differences”. Very useful when you want to see exactly what changes have been made.
Export	This command produces an compressed archive of all versioned files (of a specific revision).
GPO	Group policy object
HEAD	HEAD is a synonym for the currently active branch (to be more precise in Git HEAD can also be so-called "detached" and directly pointing to a commit instead of a branch).
History	Show the revision history of a file or folder. Also known as “Log”.
Log	Show the revision history of a file or folder. Also known as “History”.

Merge

The process by which changes from the repository are added to your working tree without disrupting any changes you have already made locally. Sometimes these changes cannot be reconciled automatically and the working tree is said to be in conflict.

Merging happens automatically when you pull changes, cherry-pick, or rebase. You can also merge specific changes from another branch using TortoiseGit's Merge command.

Patch

If a working tree has changes to text files only, it is possible to use Git's Diff command to generate a single file summary of those changes in Unified Diff format. A file of this type is often referred to as a “Patch”, and it can be emailed to someone else (or to a mailing list) and applied to another working tree. Someone without commit access can make changes and submit a patch file for an authorized committer to apply. Or if you are unsure about a change you can submit a patch for others to review.

Pull

This Git command pulls down the latest changes from the repository into your working tree, merging any changes made by others with local changes in the working tree.

Repository

A repository is a place where data is stored and maintained. A repository can be a place where multiple databases or files are located for distribution over a network, or a repository can be a location that is directly accessible to the user without having to travel across a network. Git is a distributed version control system - each working tree contains its own repository (in the `.git` folder). A Git repository does not require network to work with most operations. Network is required only when you need to synchronize changes with remote repositories.

Resolve

When files in a working tree are left in a conflicted state following a merge, those conflicts must be sorted out by a human using an editor (or perhaps TortoiseGitMerge). This process is referred to as “Resolving Conflicts”. When this is complete you can mark the conflicted files as being resolved, which allows them to be committed.

Revert

If you have made changes and decide you want to undo them, you can use the “revert” command to go back to the version from HEAD.

Revision

Every time you commit a set of changes, you create one new “revision” in the repository. Each revision represents the state of the repository tree at a certain point in its history. If you want to go back in time you can examine the repository as it was at a specific revision.

In another sense, a revision can refer to the set of changes that were made when that revision was created.

SVN

A frequently-used abbreviation for Subversion.

TortoiseGit provides git-svn interoperability. You can fetch partial or whole history from an SVN remote and store as a local git repository. This allows you to browse the history and create commits locally. You can finally commit your changes to an SVN remote.

Switch/Checkout

Updates all files in the working tree to a specific version. This is normally used for switching/checking out branches.

Update

The corresponding command for the SVN update command is *Pull*.

Working Copy

See “Working Tree”.

Working Tree	This is your local “sandbox”, the area where you work on the versioned files, and it normally resides on your local hard disk. You create a working tree by doing a “Clone” of a repository, and you feed your changes back into the repository using “Commit”.

Index

A
add, 57
annotate, 84
authentication, 8
automation, 159, 162

B
Bisect, 69
blame, 84
branch, 70
Browse All Refs, 28
bug tracker, 87
bug tracking, 87
bugtracker, 87

C
case change, 63
changelist, 32
check in, 12
check new version, 154
checkout, 10
Cherry pick, 74
cleanup, 65
client hooks, 122
Clone Repository, 9
COM, 139, 149
COM GitWCRev interface, 143
command line, 159, 162
commit, 12
commit messages, 34
compare, 51
compare revisions, 52
conflict, 76
context menu, 5
context menu entries, 156
copy files, 62
Create Repository, 9
create working tree, 10
Cygwin Git, 90

D
daemon, 28
Daemon, 28
dcommit, 136
delete, 62
Delete remote tags, 28
deploy, 154
dictionary, 3
diff, 51, 79
diff tools, 57
diffing, 22
disable functions, 156
domain controller, 154
drag handler, 7
drag-n-drop, 7

E
explorer, 1
export, 86
export changes, 52
Extern DLL Path, 90
Extra PATH, 90

F
FAQ, 148
Fetch, 23
filter, 41

G
git(1), 275
git-add(1), 301
git-am(1), 307
git-annotate(1), 311
git-apply(1), 314
git-archimport(1), 319
git-archive(1), 320
git-bisect(1), 324
git-blame(1), 330
git-branch(1), 337
git-bugreport(1), 344
git-bundle(1), 345
git-cat-file(1), 349
git-check-attr(1), 354
git-check-ignore(1), 356
git-check-mailmap(1), 358
git-check-ref-format(1), 359
git-checkout(1), 363
git-checkout-index(1), 360
git-cherry(1), 375
git-cherry-pick(1), 371
git-citool(1), 377
git-clean(1), 378
git-clone(1), 380
git-column(1), 387
git-commit(1), 394
git-commit-graph(1), 389
git-commit-tree(1), 391
git-config(1), 403
git-count-objects(1), 499
git-credential(1), 499
git-credential-cache(1), 502
git-credential-cache--daemon(1), 502
git-credential-store(1), 503
git-cvsexportcommit(1), 505
git-cvsimport(1), 507
git-cvsserver(1), 510
git-daemon(1), 516
git-describe(1), 520
git-diagnose(1), 523
git-diff(1), 586
git-diff-files(1), 524

git-diff-index(1), 541
git-diff-tree(1), 558
git-difftool(1), 610
git-fast-export(1), 612
git-fast-import(1), 616
git-fetch(1), 638
git-fetch-pack(1), 636
git-filter-branch(1), 652
git-fmt-merge-msg(1), 660
git-for-each-ref(1), 662
git-for-each-repo(1), 669
git-format-patch(1), 670
git-fsck(1), 687
git-fsck-objects(1), 686
git-fsmonitor--daemon(1), 694
git-gc(1), 695
git-get-tar-commit-id(1), 700
git-grep(1), 700
git-gui(1), 706
git-hash-object(1), 707
git-help(1), 708
git-hook(1), 711
git-http-backend(1), 712
git-http-fetch(1), 716
git-http-push(1), 717
git-imap-send(1), 719
git-index-pack(1), 722
git-init(1), 724
git-init-db(1), 724
git-instaweb(1), 727
git-interpret-trailers(1), 728
git-log(1), 735
git-ls-files(1), 778
git-ls-remote(1), 783
git-ls-tree(1), 785
git-mailinfo(1), 788
git-mailsplit(1), 790
git-maintenance(1), 791
git-merge(1), 808
git-merge-base(1), 797
git-merge-file(1), 800
git-merge-index(1), 802
git-merge-one-file(1), 803
git-merge-tree(1), 803
git-mergetool(1), 822
git-mergetool--lib(1), 821
git-mktag(1), 828
git-mktree(1), 829
git-multi-pack-index(1), 830
git-mv(1), 829
git-name-rev(1), 832
git-notes(1), 834
git-p4(1), 840
git-pack-objects(1), 851
git-pack-redundant(1), 857
git-pack-refs(1), 858
git-patch-id(1), 859
git-prune(1), 861
git-prune-packed(1), 860
git-pull(1), 862
git-push(1), 876
git-quiltimport(1), 890
git-range-diff(1), 891
git-read-tree(1), 895
git-rebase(1), 900
git-receive-pack(1), 921
git-reflog(1), 925
git-remote(1), 929
git-remote-ext(1), 926
git-remote-fd(1), 928
git-repack(1), 933
git-replace(1), 936
git-replay(1), 939
git-request-pull(1), 954
git-rerere(1), 955
git-reset(1), 958
git-restore(1), 964
git-rev-list(1), 967
git-rev-parse(1), 994
git-revert(1), 1007
git-rm(1), 1010
git-send-email(1), 1012
git-send-pack(1), 1021
git-sh-i18n(1), 1024
git-sh-i18n--envsubst(1), 1024
git-sh-setup(1), 1025
git-shell(1), 1026
git-shortlog(1), 1028
git-show(1), 1045
git-show-branch(1), 1039
git-show-index(1), 1042
git-show-ref(1), 1043
git-sparse-checkout(1), 1071
git-stage(1), 1077
git-stash(1), 1077
git-status(1), 1083
git-stripspace(1), 1090
git-submodule(1), 1096
git-svn(1), 1102
git-switch(1), 1092
git-symbolic-ref(1), 1118
git-tag(1), 1119
git-unpack-file(1), 1124
git-unpack-objects(1), 1125
git-update-index(1), 1126
git-update-ref(1), 1133
git-update-server-info(1), 1136
git-upload-archive(1), 1137
git-upload-pack(1), 1137
git-var(1), 1139
git-verify-commit(1), 1140
git-verify-pack(1), 1141
git-verify-tag(1), 1142
git-version(1), 1142
git-web--browse(1), 1143
git-whatchanged(1), 1145

git-worktree(1), 1145
git-write-tree(1), 1152
git.exe path, 90
gitattributes(5), 1158
gitcli(7), 1155
gitcore-tutorial(7), 246
gitcredentials(7), 1174
gitcvs-migration(7), 266
gitdiffcore(7), 1178
giteveryday(7), 268
gitfaq(7), 1185
gitformat-bundle(5), 1270
gitformat-chunk(5), 1272
gitformat-commit-graph(5), 1273
gitformat-index(5), 1277
gitformat-pack(5), 1283
gitformat-signature(5), 1293
gitglossary(7), 1260
githooks(5), 1191
gitignore(5), 1182
gitk(1), 1200
gitmailmap(5), 1203
gitmodules(5), 1205
gitnamespaces(7), 1207
gitprotocol-capabilities(5), 1296
gitprotocol-common(5), 1301
gitprotocol-http(5), 1303
gitprotocol-pack(5), 1310
gitprotocol-v2(5), 1320
gitremote-helpers(7), 1208
gitrepository-layout(5), 1215
gitrevisions(7), 1221
gitsubmodules(7), 1227
gittutorial(7), 233
gittutorial-2(7), 241
GitWCRev, 139
gitweb(1), 1231
gitweb.conf(5), 1241
gitworkflows(7), 1254
globbing, 60
GPG signing, 70
GPO, 154
graph, 46
group policies, 154, 156

H

history, 34
hook scripts, 122

I

IBugTraqProvider, 149
icons, 18
ignore, 59
image diff, 55
install, 3
issue tracker, 87, 149

L

language packs, 3
LFS, 136
Lock, 136
log, 34
log messages, 34
log navigation, 43

M

mark release, 70
maximize, 9
merge, 72
merge tools, 57
modifications, 20
move, 62
msi, 154
MSYS2 Git, 90

O

overlay priority, 164
overlays, 18, 164

P

patch, 79
pattern matching, 60
plugin, 149
praise, 84
proxy server, 108
Pull, 22
pull request, 79
Push, 25

R

Rebase, 75
Reference Browser, 28
RefLog, 48
refs, 28
registry, 130
remove, 62
rename, 62
repo-browser, 49
Repository, 9, 9
request pull, 79
Reset, 66
resolve, 76
revert, 64
revision, 46
revision graph, 46
revision log dialog, 34
right drag, 7
right-click, 5

S

scalar(1), 1153
send changes, 12
settings, 90

spell checker, 3
stash, 68
statistics, 44
status, 18, 20
Submodule Diff Dialog, 54
SUBST drives, 106
SubWCRev, 139
svn, 136
svn commit, 136
Switch, 10
Sync, 26

T

tag, 70
TortoiseGitIDiff, 55
translations, 3

U

undo, 64
unified diff, 79
unified-diff viewer, 113
unversioned 'working tree', 86
unversioned files/folders, 59
upgrade check, 154

V

version, 154
version control, 1
version extraction, 139
version new files, 57
version number in files, 139
view changes, 18

W

windows properties, 19
Windows shell, 1
working tree status, 18
worktree, 136