Git - the stupid content tracker

"git" can mean anything, depending on your mood.

- random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.
- stupid. contemptible and despicable. simple. Take your pick from the dictionary of slang.
- "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- "goddamn idiotic truckload of sh*t": when it breaks

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.

Git is an Open Source project covered by the GNU General Public License version 2 (some parts of it are under different licenses, compatible with the GPLv2). It was originally written by Linus Torvalds with help of a group of hackers around the net.

Please read the file INSTALL for installation instructions.

See Documentation/gittutorial.txt to get started, then see Documentation/giteveryday.txt for a useful minimum set of commands, and Documentation/git-commandname.txt for documentation of each command. If git has been correctly installed, then the tutorial can also be read with "man gittutorial" or "git help tutorial", and the documentation of each command with "man git-commandname" or "git help commandname".

CVS users may also want to read Documentation/gitcvs-migration.txt ("man gitcvs-migration" or "git help cvs-migration" if git is installed).

Many Git online resources are accessible from http://git-scm.com/ including full documentation and Git related tools.

The user discussion and development of Git take place on the Git mailing list — everyone is welcome to post bug reports, feature requests, comments and patches to git@vger.kernel.org (read Documentation/SubmittingPatches for instructions on patch submission). To subscribe to the list, send an email with just "subscribe git" in the body to majordomo@vger.kernel.org. The mailing list archives are available at http://news.gmane.org/gmane.comp.version-control.git/, http://marc.info/?l=git and other archival sites.

The maintainer frequently sends the "What's cooking" reports that list the current status of various development topics to the mailing list. The discussion following them give a good reference for project status, development direction and remaining tasks.

Last updated 2015-05-03 21:16:44 CEST

gittutorial(7) Manual Page

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.

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 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:
Your branch is up-to-date with 'origin/master'.
  (use "git reset HEAD <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 (less than 50 character) 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, 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 checkout 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 checkout 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 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 rsync or http; see 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 git-push(1) and 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 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 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:

- git-format-patch(1), 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.
- 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.
- gitworkflows(7): Gives an overview of recommended workflows.
- giteveryday(7): Everyday Git with 20 Commands Or So.
- gitcvs-migration(7): Git for CVS users.

SEE ALSO

gittutorial-2(7), gitcvs-migration(7), gitcore-tutorial(7), gitglossary(7), git-help(1), gitworkflows(7), giteveryday(7), The Git User's Manual

GIT

Part of the git(1) suite.

Last updated 2014-12-13 19:39:10 CET

gittutorial-2(7) Manual Page

NAME

gittutorial-2 - A tutorial introduction to Git: part two

SYNOPSIS

> git *

DESCRIPTION

You should work through 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 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 reset HEAD <file>..." to unstage)

        new file:   closing.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <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 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 giteveryday(7). You should be able to find any unknown jargon in gitglossary(7).

The Git User’s Manual provides a more comprehensive introduction to Git.

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.

For Git developers, gitcore-tutorial(7) goes into detail on the lower-level Git mechanisms involved in, for example, creating a new commit.

SEE ALSO

gittutorial(7), gitcvs-migration(7), gitcore-tutorial(7), gitglossary(7), git-help(1), giteveryday(7), The Git User’s Manual

GIT

Part of the git(1) suite.

Last updated 2014-12-13 19:39:10 CET

giteveryday(7) Manual Page

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 command 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.

- git-init(1) to create a new repository.
- git-log(1) to see what happened.
- git-checkout(1) and git-branch(1) to switch branches.
- git-add(1) to manage the index file.
- git-diff(1) and git-status(1) to see what you are in the middle of doing.
- git-commit(1) to advance the current branch.
- git-reset(1) and git-checkout(1) (with pathname parameters) to undo changes.

- git-merge(1) to merge between local branches.
- git-rebase(1) to maintain topic branches.
- git-tag(1) to mark a known point.

Examples

Use a tarball as a starting point for a new repository.

```
$ tar zxf frotz.tar.gz
$ cd frotz
$ git init
$ git add . <1>
$ git commit -m "import of frotz source tree."
$ git tag v2.43 <2>
```

1. add everything under the current directory.
2. make a lightweight, unannotated tag.

Create a topic branch and develop.

```
$ git checkout -b alsa-audio <1>
$ edit/compile/test
$ git checkout -- curses/ux_audio_oss.c <2>
$ git add curses/ux_audio_alsa.c <3>
$ edit/compile/test
$ git diff HEAD <4>
$ git commit -a -s <5>
$ edit/compile/test
$ git diff HEAD^ <6>
$ git commit -a --amend <7>
$ git checkout master <8>
$ git merge alsa-audio <9>
$ git log --since='3 days ago' <10>
$ git log v2.43.. curses/ <11>
```

1. create a new topic branch.
2. revert your botched changes in `curses/ux_audio_oss.c`.
3. you need to tell Git if you added a new file; removal and modification will be caught if you do `git commit -a` later.
4. to see what changes you are committing.
5. commit everything, as you have tested, with your sign-off.
6. look at all your changes including the previous commit.
7. amend the previous commit, adding all your new changes, using your original message.
8. switch to the master branch.
9. merge a topic branch into your master branch.
10. 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.
11. 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.

- git-clone(1) from the upstream to prime your local repository.
- git-pull(1) and git-fetch(1) from "origin" to keep up-to-date with the upstream.
- git-push(1) to shared repository, if you adopt CVS style shared repository workflow.
- git-format-patch(1) to prepare e-mail submission, if you adopt Linux kernel-style public forum workflow.
- git-send-email(1) to send your e-mail submission without corruption by your MUA.
- git-request-pull(1) to create a summary of changes for your upstream to pull.

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 checkout -b mine master <1>
$ edit/compile/test; git commit -a -s <2>
$ git format-patch master <3>
$ git send-email --to="person <email@example.com>" 00*.patch <4>
$ git checkout master <5>
$ git pull <6>
$ git log -p ORIG_HEAD.. arch/i386 include/asm-i386 <7>
$ git ls-remote --heads http://git.kernel.org/.../jgarzik/libata-dev.git <8>
$ git pull git://git.kernel.org/pub/.../jgarzik/libata-dev.git ALL <9>
$ git reset --hard ORIG_HEAD <10>
$ git gc <11>
```

1. checkout a new branch `mine` from master.
2. repeat as needed.
3. extract patches from your branch, relative to master,
4. and email them.
5. return to `master`, ready to see what's new
6. `git pull` fetches from `origin` by default and merges into the current branch.
7. immediately after pulling, look at the changes done upstream since last time we checked, only in the area we are interested in.
8. check the branch names in an external repository (if not known).
9. fetch from a specific branch `ALL` from a specific repository and merge it.
10. revert the pull.
11. garbage collect leftover objects from reverted pull.

Push into another repository.

```
satellite$ git clone mothership:frotz frotz <1>
satellite$ cd frotz
satellite$ git config --get-regexp '^(remote|branch)\.' <2>
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/* <3>
satellite$ edit/compile/test/commit
satellite$ git push origin <4>

mothership$ cd frotz
mothership$ git checkout master
mothership$ git merge satellite/master <5>
```

1. mothership machine has a frotz repository under your home directory; clone from it to start a repository on the satellite machine.
2. 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.
3. arrange `git push` to push all local branches to their corresponding branch of the mothership machine.
4. 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).
5. on mothership machine, merge the work done on the satellite machine into the master branch.

Branch off of a specific tag.

```
$ git checkout -b private2.6.14 v2.6.14 <1>
$ edit/compile/test; git commit -a
$ git checkout master
$ git cherry-pick v2.6.14..private2.6.14 <2>
```

1. create a private branch based on a well known (but somewhat behind) tag.
2. forward port all changes in `private2.6.14` branch to `master` branch without a formal "merging". Or longhand

   ```
   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. An sub-area lieutenant for a repository will act both as a participant and as an integrator.

- git-am(1) to apply patches e-mailed in from your contributors.
- git-pull(1) to merge from your trusted lieutenants.
- git-format-patch(1) to prepare and send suggested alternative to contributors.
- git-revert(1) to undo botched commits.
- git-push(1) to publish the bleeding edge.

Examples

A typical integrator's Git day.

```
$ git status <1>
$ git branch --no-merged master <2>
$ mailx <3>
& s 2 3 4 5 ./+to-apply
& s 7 8 ./+hold-linus
& q
$ git checkout -b topic/one master
$ git am -3 -i -s ./+to-apply <4>
$ compile/test
$ git checkout -b hold/linus && git am -3 -i -s ./+hold-linus <5>
$ git checkout topic/one && git rebase master <6>
$ git checkout pu && git reset --hard next <7>
$ git merge topic/one topic/two && git merge hold/linus <8>
$ git checkout maint
$ git cherry-pick master~4 <9>
$ compile/test
$ git tag -s -m "GIT 0.99.9x" v0.99.9x <10>
$ git fetch ko && for branch in master maint next pu <11>
    do
        git show-branch ko/$branch $branch <12>
    done
$ git push --follow-tags ko <13>
```

1. see what you were in the middle of doing, if anything.
2. see which branches haven't been merged into `master` yet. Likewise for any other integration branches e.g. `maint`, `next` and `pu` (potential updates).
3. read mails, save ones that are applicable, and save others that are not quite ready (other mail readers are available).
4. apply them, interactively, with your sign-offs.
5. create topic branch as needed and apply, again with sign-offs.
6. rebase internal topic branch that has not been merged to the master or exposed as a part of a stable branch.
7. restart `pu` every time from the next.
8. and bundle topic branches still cooking.
9. backport a critical fix.
10. create a signed tag.
11. make sure master was not accidentally rewound beyond that already pushed out. `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/pu
            push = refs/heads/maint
    ```

12. In the output from `git show-branch`, `master` should have everything `ko/master` has, and `next` should have everything `ko/next` has, etc.
13. push out the bleeding edge, together with new tags that point into the pushed history.

Repository Administration

A repository administrator uses the following tools to set up and maintain access to the repository by developers.

- git-daemon(1) to allow anonymous download from repository.
- git-shell(1) can be used as a *restricted login shell* for shared central repository users.
- git-http-backend(1) provides a server side implementation of Git-over-HTTP ("Smart http") allowing both fetch and push services.
- gitweb(1) provides a web front-end to Git repositories, which can be set-up using the git-instaweb(1) script.

update hook howto 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.

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 <1>
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 <2>
/usr/bin/git-shell
```

1. 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.
2. in many distributions /etc/shells needs to list what is used as the login shell.

CVS-style shared repository.

```
$ grep git /etc/group <1>
git:x:9418:alice,bob,cindy,david
$ cd /home/devo.git
$ ls -l <2>
  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 <3>
  -r-xr-xr-x    1 david git  3536 Dec  4 22:40 update
$ cat info/allowed-users <4>
refs/heads/master       alice\|cindy
```

```
refs/heads/doc-update   bob
refs/tags/v[0-9]*       david
```

1. place the developers into the same git group.
2. and make the shared repository writable by the group.
3. use update-hook example by Carl from Documentation/howto/ for branch policy control.
4. 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 git(1) suite

Last updated 2014-11-27 19:58:08 CET

gitworkflows(7) Manual Page

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 git-blame(1) and 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 save --keep-index` to run the test suite independent of other uncommitted changes; see the EXAMPLES section of git-stash(1).

MANAGING BRANCHES

There are two main tools that can be used to include changes from one branch on another: git-merge(1) and 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.

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:

- *pu* (proposed updates) 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 *pu*), and "graduates" to *master* for the next release once it is considered stable enough.

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:

Rule: Merge upwards

> Always commit your fixes to the oldest supported branch that require them. Then (periodically) merge the integration branches upwards into each other.

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 git-cherry-pick(1)) downwards. This will happen a few times and is nothing to worry about unless you do it very frequently.

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:

Rule: Topic branches

> 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.

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 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 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:

Rule: Merge to downstream only at well-defined points

> 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.

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.

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.

Rule: Throw-away integration branches

> To test the interaction of several topics, merge them into a throw-away branch. You must never base any work on such a branch!

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 *pu*.

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:

Recipe: Verify ***master*** is a superset of ***maint***

```
git log master..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:

Recipe: Release tagging

```
git tag -s -m "Git X.Y.Z" vX.Y.Z master
```

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*.

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).

Recipe: Copy maint

```
git branch maint-X.Y.(Z-1) 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:

Recipe: Update maint to new release

- `git checkout maint`

- `git merge --ff-only master`

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.

Branch management for next and pu 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*:

Recipe: Rewind and rebuild next

- `git checkout next`
- `git reset --hard master`
- `git merge ai/topic_in_next1`
- `git merge ai/topic_in_next2`
- …

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 *pu*. A public announcement is not necessary since *pu* 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.

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:

- git-push(1) copies your branches to a remote repository, usually to one that can be read by all involved parties;
- git-fetch(1) that copies remote branches to your repository; and
- 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:

Recipe: Push/pull: Publishing branches/topics

`git push <remote> <branch>` and tell everyone where they can fetch from.

You will still have to tell people by other means, such as mail. (Git provides the 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:

Recipe: Push/pull: Staying up to date

Use `git fetch <remote>` or `git remote update` to stay 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.

Recipe: Push/pull: Merging remote topics

```
git pull <url> <branch>
```

Occasionally, the maintainer may get merge conflicts when he tries to pull changes from downstream. In this case, he 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.

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 git-format-patch(1) to generate the corresponding emails (highly recommended over manually formatting them because it makes the maintainer’s life easier).

Recipe: format-patch/am: Publishing branches/topics

- `git format-patch -M upstream..topic` to turn them into preformatted patch files
- `git send-email --to=<recipient> <patches>`

See the git-format-patch(1) and 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):

Recipe: format-patch/am: Keeping topics up to date

```
git pull --rebase <url> <branch>
```

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:

Recipe: format-patch/am: Importing patches

```
git am < patch
```

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 git-am(1) for other options.

SEE ALSO

gittutorial(7), git-push(1), git-pull(1), git-merge(1), git-rebase(1), git-format-patch(1), git-send-email(1), git-am(1)

GIT

Part of the git(1) suite.

Last updated 2014-11-27 19:54:14 CET

git(1) Manual Page

NAME

git - the stupid content tracker

SYNOPSIS

```
git [--version] [--help] [-C <path>] [-c <name>=<value>]
    [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
    [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
    [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
    <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 gittutorial(7) to get started, then see giteveryday(7) for a useful minimum set of commands. The Git User's Manual 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". gitcli(7) manual page gives you an overview of the command-line command syntax.

Formatted and hyperlinked version of the latest Git documentation can be viewed at `http://git-htmldocs.googlecode.com/git/git.html`.

OPTIONS

--version
: Prints the Git suite version that the *git* program came from.

--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 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>`.

  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.

--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).

--no-pager
: Do not pipe Git output into a pager.

--git-dir=<path>
: Set the path to the repository. 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.

--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 git-config(1) for a more detailed discussion).

--namespace=<path>
: Set the Git namespace. See 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. See git-replace(1) for more information.

--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`.

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.

Main porcelain commands

git-add(1)
: Add file contents to the index.

git-am(1)
: Apply a series of patches from a mailbox.

git-archive(1)
: Create an archive of files from a named tree.

git-bisect(1)
: Find by binary search the change that introduced a bug.

git-branch(1)
: List, create, or delete branches.

git-bundle(1)
: Move objects and refs by archive.

git-checkout(1)

Checkout a branch or paths to the working tree.

git-cherry-pick(1)
: Apply the changes introduced by some existing commits.

git-citool(1)
: Graphical alternative to git-commit.

git-clean(1)
: Remove untracked files from the working tree.

git-clone(1)
: Clone a repository into a new directory.

git-commit(1)
: Record changes to the repository.

git-describe(1)
: Show the most recent tag that is reachable from a commit.

git-diff(1)
: Show changes between commits, commit and working tree, etc.

git-fetch(1)
: Download objects and refs from another repository.

git-format-patch(1)
: Prepare patches for e-mail submission.

git-gc(1)
: Cleanup unnecessary files and optimize the local repository.

git-grep(1)
: Print lines matching a pattern.

git-gui(1)
: A portable graphical interface to Git.

git-init(1)
: Create an empty Git repository or reinitialize an existing one.

git-log(1)
: Show commit logs.

git-merge(1)
: Join two or more development histories together.

git-mv(1)
: Move or rename a file, a directory, or a symlink.

git-notes(1)
: Add or inspect object notes.

git-pull(1)
: Fetch from and integrate with another repository or a local branch.

git-push(1)
: Update remote refs along with associated objects.

git-rebase(1)
: Forward-port local commits to the updated upstream head.

git-reset(1)
: Reset current HEAD to the specified state.

git-revert(1)
: Revert some existing commits.

git-rm(1)
: Remove files from the working tree and from the index.

git-shortlog(1)
: Summarize *git log* output.

git-show(1)
: Show various types of objects.

git-stash(1)
: Stash the changes in a dirty working directory away.

git-status(1)
: Show the working tree status.

git-submodule(1)
: Initialize, update or inspect submodules.

git-tag(1)
: Create, list, delete or verify a tag object signed with GPG.

gitk(1)
: The Git repository browser.

Ancillary Commands

Manipulators:

git-config(1)
: Get and set repository or global options.

git-fast-export(1)
: Git data exporter.

git-fast-import(1)
: Backend for fast Git data importers.

git-filter-branch(1)
: Rewrite branches.

git-mergetool(1)
: Run merge conflict resolution tools to resolve merge conflicts.

git-pack-refs(1)
: Pack heads and tags for efficient repository access.

git-prune(1)
: Prune all unreachable objects from the object database.

git-reflog(1)
: Manage reflog information.

git-relink(1)
: Hardlink common objects in local repositories.

git-remote(1)
: Manage set of tracked repositories.

git-repack(1)
: Pack unpacked objects in a repository.

git-replace(1)
: Create, list, delete refs to replace objects.

Interrogators:

git-annotate(1)
: Annotate file lines with commit information.

git-blame(1)
: Show what revision and author last modified each line of a file.

git-cherry(1)
: Find commits yet to be applied to upstream.

git-count-objects(1)
: Count unpacked number of objects and their disk consumption.

git-difftool(1)
: Show changes using common diff tools.

git-fsck(1)
: Verifies the connectivity and validity of the objects in the database.

git-get-tar-commit-id(1)
: Extract commit ID from an archive created using git-archive.

git-help(1)
: Display help information about Git.

git-instaweb(1)
: Instantly browse your working repository in gitweb.

git-merge-tree(1)
: Show three-way merge without touching index.

git-rerere(1)
: Reuse recorded resolution of conflicted merges.

git-rev-parse(1)
: Pick out and massage parameters.

git-show-branch(1)
: Show branches and their commits.

git-verify-commit(1)

Check the GPG signature of commits.

git-verify-tag(1)
: Check the GPG signature of tags.

git-whatchanged(1)
: Show logs with difference each commit introduces.

gitweb(1)
: Git web interface (web frontend to Git repositories).

Interacting with Others

These commands are to interact with foreign SCM and with other people via patch over e-mail.

git-archimport(1)
: Import an Arch repository into Git.

git-cvsexportcommit(1)
: Export a single commit to a CVS checkout.

git-cvsimport(1)
: Salvage your data out of another SCM people love to hate.

git-cvsserver(1)
: A CVS server emulator for Git.

git-imap-send(1)
: Send a collection of patches from stdin to an IMAP folder.

git-p4(1)
: Import from and submit to Perforce repositories.

git-quiltimport(1)
: Applies a quilt patchset onto the current branch.

git-request-pull(1)
: Generates a summary of pending changes.

git-send-email(1)
: Send a collection of patches as emails.

git-svn(1)
: Bidirectional operation between a Subversion repository and Git.

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 git-update-index(1) and 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.

Manipulation commands

git-apply(1)
: Apply a patch to files and/or to the index.

git-checkout-index(1)
: Copy files from the index to the working tree.

git-commit-tree(1)
: Create a new commit object.

git-hash-object(1)
: Compute object ID and optionally creates a blob from a file.

git-index-pack(1)
: Build pack index file for an existing packed archive.

git-merge-file(1)
: Run a three-way file merge.

git-merge-index(1)
: Run a merge for files needing merging.

git-mktag(1)
: Creates a tag object.

git-mktree(1)
: Build a tree-object from ls-tree formatted text.

git-pack-objects(1)
: Create a packed archive of objects.

git-prune-packed(1)
: Remove extra objects that are already in pack files.

git-read-tree(1)
: Reads tree information into the index.

git-symbolic-ref(1)
: Read, modify and delete symbolic refs.

git-unpack-objects(1)
: Unpack objects from a packed archive.

git-update-index(1)
: Register file contents in the working tree to the index.

git-update-ref(1)
: Update the object name stored in a ref safely.

git-write-tree(1)
: Create a tree object from the current index.

Interrogation commands

git-cat-file(1)
: Provide content or type and size information for repository objects.

git-diff-files(1)
: Compares files in the working tree and the index.

git-diff-index(1)
: Compare a tree to the working tree or index.

git-diff-tree(1)
: Compares the content and mode of blobs found via two tree objects.

git-for-each-ref(1)
: Output information on each ref.

git-ls-files(1)
: Show information about files in the index and the working tree.

git-ls-remote(1)
: List references in a remote repository.

git-ls-tree(1)
: List the contents of a tree object.

git-merge-base(1)
: Find as good common ancestors as possible for a merge.

git-name-rev(1)
: Find symbolic names for given revs.

git-pack-redundant(1)
: Find redundant pack files.

git-rev-list(1)
: Lists commit objects in reverse chronological order.

git-show-index(1)
: Show packed archive index.

git-show-ref(1)
: List references in a local repository.

git-unpack-file(1)
: Creates a temporary file with a blob’s contents.

git-var(1)
: Show a Git logical variable.

git-verify-pack(1)
: Validate packed Git archive files.

In general, the interrogate commands do not touch the files in the working tree.

Synching repositories

git-daemon(1)
: A really simple server for Git repositories.

git-fetch-pack(1)
: Receive missing objects from another repository.

git-http-backend(1)
: Server side implementation of Git over HTTP.

git-send-pack(1)
: Push objects over Git protocol to another repository.

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.

git-http-fetch(1)
: Download from a remote Git repository via HTTP.

git-http-push(1)
: Push objects over HTTP/DAV to another repository.

git-parse-remote(1)
: Routines to help parsing remote repository access parameters.

git-receive-pack(1)
: Receive what is pushed into the repository.

git-shell(1)
: Restricted login shell for Git-only SSH access.

git-upload-archive(1)
: Send archive back to git-archive.

git-upload-pack(1)
: Send objects packed back to git-fetch-pack.

Internal helper commands

These are internal helper commands used by other commands; end users typically do not use them directly.

git-check-attr(1)
: Display gitattributes information.

git-check-ignore(1)
: Debug gitignore / exclude files.

git-check-mailmap(1)
: Show canonical names and email addresses of contacts.

git-check-ref-format(1)
: Ensures that a reference name is well formed.

git-column(1)
: Display data in columns.

git-credential(1)
: Retrieve and store user credentials.

git-credential-cache(1)
: Helper to temporarily store passwords in memory.

git-credential-store(1)
: Helper to store credentials on disk.

git-fmt-merge-msg(1)
: Produce a merge commit message.

git-interpret-trailers(1)
: help add structured information into commit messages.

git-mailinfo(1)
: Extracts patch and authorship from a single e-mail message.

git-mailsplit(1)
: Simple UNIX mbox splitter program.

git-merge-one-file(1)
: The standard helper program to use with git-merge-index.

git-patch-id(1)
: Compute unique ID for a patch.

git-sh-i18n(1)
: Git's i18n setup code for shell scripts.

git-sh-setup(1)
: Common Git shell script setup code.

git-stripspace(1)
: Remove unnecessary whitespace.

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 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 gitrevisions(7).

File/Directory Structure

Please see the gitrepository-layout(5) document.

Read githooks(5) for more details about each hook.

Higher level SCMs may provide and manage additional information in the `$GIT_DIR`.

Terminology

Please see gitglossary(7).

Environment Variables

Various Git commands use the following environment variables:

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 Cogito etc.

GIT_INDEX_FILE
: This environment allows the specification of an alternate index file. If not specified, the default of `$GIT_DIR/index` is used.

GIT_INDEX_VERSION
: This environment variable allows the specification of an index version for new repositories. It won't affect existing index files. By default index file version 2 or 3 is used. See 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.

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 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. Normally, 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 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 Commits

GIT_AUTHOR_NAME

GIT_AUTHOR_EMAIL

GIT_AUTHOR_DATE

GIT_COMMITTER_NAME

GIT_COMMITTER_EMAIL

GIT_COMMITTER_DATE

EMAIL

see git-commit-tree(1)

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, instead of the diff invocation described above. 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.

other

GIT_MERGE_VERBOSITY

A number controlling the amount of output shown by the recursive merge strategy. Overrides merge.verbosity. See 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 git-config(1).

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 git-var(1) and the `core.editor` option in 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 will be given exactly two or four arguments: the *username@host* (or just *host*) from the URL and the shell command to execute on that remote system, optionally preceded by *-p* (literally) and the *port* from the URL when it specifies something other than the default SSH port.

`$GIT_SSH_COMMAND` takes precedence over `$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_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 git-config(1).

GIT_TERMINAL_PROMPT
: If this environment variable is set to `0`, git will not prompt on the terminal (e.g., when asking for HTTP authentication).

GIT_CONFIG_NOSYSTEM
: Whether to skip reading settings from the system-wide `$(prefix)/etc/gitconfig` file. This environment variable can be used along with `$HOME` and `$XDG_CONFIG_HOME` to create a predictable environment for a picky script, or you can set it temporarily to avoid using a buggy `/etc/gitconfig` file while waiting for someone with sufficient permissions to fix it.

GIT_FLUSH
: If this environment variable is set to "1", 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 "0", 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 write the trace messages into it.

  Unsetting the variable, or setting it to empty, "0" or "false" (case insensitive) disables trace messages.

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". See *GIT_TRACE* for available trace output options.

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_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_LITERAL_PATHSPECS
: Setting this variable to `1` 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 variable to `1` will cause Git to treat all pathspecs as glob patterns (aka "glob" magic).

GIT_NOGLOB_PATHSPECS
: Setting this variable to `1` will cause Git to treat all pathspecs as literal (aka "literal" magic).

GIT_ICASE_PATHSPECS
: Setting this variable to `1` will cause Git to treat all pathspecs as case-insensitive.

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 set to `1`, include broken or badly named refs when iterating over lists of refs. In a normal, non-corrupted repository, this does nothing. However, enabling it may help git to detect and abort some operations in the presence of broken refs. Git sets this variable automatically when performing destructive operations like git-prune(1). You should not need to set it yourself unless you want to be paranoid about making sure an operation has touched every ref (e.g., because you are cloning a repository to make a backup).

Discussion

More detail on the following is available from the Git concepts chapter of the user-manual and 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. Refs with names beginning `ref/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 `ref/tags/`. A special 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.

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 and gitcore-tutorial(7) both provide introductions to the underlying Git architecture.

See gitworkflows(7) for an overview of recommended workflows.

See also the howto documents for some useful examples.

The internals are documented in the Git API documentation.

Users migrating from CVS may also want to read 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>. http://www.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 git-shortlog(1) and 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> where the development and maintenance is primarily done. You do not have to be subscribed to the list to send a message there.

SEE ALSO

gittutorial(7), gittutorial-2(7), giteveryday(7), gitcvs-migration(7), gitglossary(7), gitcore-tutorial(7), gitcli(7), The Git User's Manual, gitworkflows(7)

GIT

Part of the git(1) suite

Last updated 2015-05-03 21:16:24 CEST

git-add(1) Manual Page

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 | --[no-]ignore-removal | [--update | -u]]
          [--intent-to-add | -N] [--refresh] [--ignore-errors] [--ignore-missing]
          [--] [<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 directory, 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 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.

-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.

-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.

--
: 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).

Configuration

The optional configuration variable `core.excludesFile` indicates a path to a file containing patterns of file names to exclude from git-add, similar to $GIT_DIR/info/exclude. Patterns in the exclude file are used in addition to those in info/exclude. See gitignore(5).

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 git-add--interactive.perl
  ```

  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, git-add--interactive.perl, 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 git-add--interactive.perl
  ```

  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
? - 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

SEE ALSO

git-status(1) git-rm(1) git-reset(1) git-mv(1) git-commit(1) git-update-index(1)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-am(1) Manual Page

NAME

git-am - Apply a series of patches from a mailbox

SYNOPSIS

```
git am [--signoff] [--keep] [--[no-]keep-cr] [--[no-]utf8]
        [--3way] [--interactive] [--committer-date-is-author-date]
        [--ignore-date] [--ignore-space-change | --ignore-whitespace]
        [--whitespace=<option>] [-C<n>] [-p<n>] [--directory=<dir>]
        [--exclude=<path>] [--include=<path>] [--reject] [-q | --quiet]
        [--[no-]scissors] [-S[<keyid>]] [--patch-format=<format>]
        [(<mbox> | <Maildir>)...]
git am (--continue | --skip | --abort)
```

DESCRIPTION

Splits mail messages in a mailbox into commit log message, authorship information and patches, and applies them to the current branch.

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:` line to the commit message, using the committer identity of yourself.

-k

--keep
: Pass `-k` flag to *git mailinfo* (see git-mailinfo(1)).

--keep-non-patch
: Pass `-b` flag to *git mailinfo* (see git-mailinfo(1)).

--[no-]keep-cr
: With `--keep-cr`, call *git mailsplit* (see 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 git-mailinfo(1)). Can be activated by default using the `mailinfo.scissors` configuration variable.

--no-scissors
: Ignore scissors lines (see git-mailinfo(1)).

-m

--message-id
: Pass the `-m` flag to *git mailinfo* (see 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 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 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 git-mailinfo(1)).

-3

--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.

--ignore-space-change

--ignore-whitespace

--whitespace=<option>

-C<n>

-p<n>

--directory=<dir>

--exclude=<path>

--include=<path>

--reject
: These flags are passed to the *git apply* (see git-apply(1)) program that applies the patch.

--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, stgit, stgit-series and hg.

-i

--interactive
: Run interactively.

--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>]
: GPG-sign commits.

--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.

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: " 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 githooks(5) for more information.

SEE ALSO

git-apply(1).

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-annotate(1) Manual Page

NAME

git-annotate - Annotate file lines with commit information

SYNOPSIS

> *git annotate* [options] file [revision]

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 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 :<regex>
: Annotate only the given line range. 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 “:<regex>” is given in place of <start> and <end>, it denotes the range from the first funcname line that matches <regex>, up to the next funcname line. “:<regex>” searches from the end of the previous `-L` range, if any, otherwise from the start of file. “^:<regex>” searches from the start of file.

-l
: Show long rev (Default: off).

-t
: Show raw timestamp (Default: off).

-S <revs-file>
: Use revisions from revs-file instead of calling git-rev-list(1).

--reverse
: 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.

-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 git-log(1) manual page.

--contents <file>

When <rev> is not specified, the command annotates the changes starting backwards from the working tree copy. This flag makes the command pretend as if the working tree copy has the contents of the named file (specify `-` to make the command read from the standard input).

--date <format>

The value is one of the following alternatives: {relative,local,default,iso,rfc,short}. 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 more information, See the discussion of the --date option at git-log(1).

-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.

-h

Show help message.

SEE ALSO

git-blame(1)

GIT

Part of the git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-apply(1) Manual Page

NAME

git-apply - Apply a patch to files and/or to the index

SYNOPSIS

```
git apply [--stat] [--numstat] [--summary] [--check] [--index] [--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] [--unsafe-paths] [<patch>...]
```

DESCRIPTION

Reads the supplied diff output (i.e. "a patch") and applies it to files. 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 git-am(1) to create commits from patches generated by 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
: When `--check` is in effect, or when applying the patch (which is the default when none of the options that disables it is in effect), make sure the patch is applicable to what the current index file records. If the file to be patched in the working tree is not up-to-date, it is flagged as an error. This flag also causes the index file to be updated.

--cached
: Apply a patch without touching the working tree. Instead take the cached data, apply the patch, and store the result in the index without using the working tree. This implies `--index`.

-3

--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, possibly leaving the conflict markers in the files in the working tree for the user to resolve. This option implies the `--index` option, and is incompatible with the `--reject` and the `--cached` options.

--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, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

-p<n>
: Remove <n> leading slashes from traditional diff paths. 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 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 applied 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 whitespace 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.

--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.

Configuration

apply.ignoreWhitespace
: Set to *change* if you want changes in whitespace to be ignored by default. Set to one of: no, none, never, false if you want changes in whitespace to be significant.

apply.whitespace
: When no `--whitespace` flag is given from the command line, this configuration item is used as the default.

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

git-am(1).

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-archimport(1) Manual Page

NAME

git-archimport - Import an 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 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 git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-archive(1) Manual Page

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 versus when given a commit ID or tag ID. In the first case the current time is used as the modification time of each file in the archive. In the latter case 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: *tar* or *zip*. If this option 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 each filename in the archive.

-o <file>

--output=<file>
: Write the archive to <file> instead of stdout.

--worktree-attributes
: Look for attributes in .gitattributes files in the working tree as well (see [ATTRIBUTES]).

<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 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

zip

-0
: Store the files instead of deflating them.

-9
: Highest and slowest compression level. You can specify any number from 1 to 9 to adjust compression speed and ratio.

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"). An output file with the same extension as `<format>` will be use this format if no other format is given.

  The "tar.gz" and "tgz" formats are defined automatically and default to `gzip -cn`. You may override them with custom commands.

tar.<format>.remote
: If true, enable `<format>` for use by remote clients via 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 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 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 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

gitattributes(5)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-bisect(1) Manual Page

NAME

git-bisect - Find by binary search the change that introduced a bug

SYNOPSIS

> *git bisect* <subcommand> <options>

DESCRIPTION

The command takes various subcommands, and different options depending on the subcommand:

```
git bisect help
git bisect start [--no-checkout] [<bad> [<good>...]] [--] [<paths>...]
git bisect bad [<rev>]
git bisect good [<rev>...]
git bisect skip [(<rev>|<range>)...]
git bisect reset [<commit>]
git bisect visualize
git bisect replay <logfile>
git bisect log
git bisect run <cmd>...
```

This command uses *git rev-list --bisect* to help drive the binary search process to find which change introduced a bug, given an old "good" commit object name and a later "bad" commit object name.

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.

Basic bisect commands: start, bad, good

Using the Linux kernel tree as an example, basic use of the bisect command is as follows:

```
$ git bisect start
$ git bisect bad                 # Current version is bad
$ git bisect good v2.6.13-rc2    # v2.6.13-rc2 was the last version
                                 # tested that was good
```

When you have specified at least one bad and one good version, the command bisects the revision tree and outputs something similar to the following:

```
Bisecting: 675 revisions left to test after this
```

The state in the middle of the set of revisions is then checked out. You would now compile that kernel and boot it. If the booted kernel works correctly, you would then issue the following command:

```
$ git bisect good                       # this one is good
```

The output of this command would be something similar to the following:

```
Bisecting: 337 revisions left to test after this
```

You keep repeating this process, compiling the tree, testing it, and depending on whether it is good or bad issuing the command "git bisect good" or "git bisect bad" to ask for the next bisection.

Eventually there will be no more revisions left to bisect, and you will have been left with the first bad kernel revision in "refs/bisect/bad".

Bisect reset

After a bisect session, to clean up the bisection state and return to the original HEAD (i.e., to quit bisecting), 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 HEAD` will leave you on the current bisection commit and avoid switching commits at all, while `git bisect reset bisect/bad` will check out the first bad revision.

Bisect visualize

To see the currently remaining suspects in *gitk*, issue the following command during the bisection process:

```
$ git bisect visualize
```

`view` may also be used as a synonym for `visualize`.

If the *DISPLAY* environment variable is not set, *git log* is used instead. You can also give command-line options such as `-p` and `--stat`.

```
$ git bisect view --stat
```

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
```

Avoiding testing a commit

If, in the middle of a bisect session, you know that the next suggested revision is not a good one to test (e.g. the change the commit introduces is known not to work in your environment and you know it does not have anything to do with the bug you are chasing), you may want to find a nearby commit and try that instead.

For example:

```
$ git bisect good/bad                   # previous round was good or bad.
Bisecting: 337 revisions left to test after this
$ 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.

Bisect skip

Instead of choosing by yourself a nearby commit, you can ask Git to do it for you by issuing the command:

```
$ git bisect skip                 # Current version cannot be tested
```

But Git may eventually be unable to tell the first bad commit among a bad commit and one or more skipped commits.

You can even skip a range of commits, instead of just one commit, using the "*<commit1>*..*<commit2>*" 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` included and `v2.6` included should be skipped.

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 path 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
```

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, and exit with a code between 1 and 127 (inclusive), except 125, if the current source code is bad.

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 a special 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.

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 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*.

SEE ALSO

Fighting regressions with git bisect, git-blame(1).

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-blame(1) Manual Page

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>]
        [--abbrev=<n>] [<rev> | --contents <file> | --reverse <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 :<regex>
: Annotate only the given line range. 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 “:<regex>” is given in place of <start> and <end>, it denotes the range from the first funcname line that matches <regex>, up to the next funcname line. “:<regex>” searches from the end of the previous `-L` range, if any, otherwise from the start of file. “^:<regex>” searches from the start of file.

-l
: Show long rev (Default: off).

-t
: Show raw timestamp (Default: off).

-S <revs-file>
: Use revisions from revs-file instead of calling git-rev-list(1).

--reverse
: 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.

-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 git-log(1) manual page.

--contents <file>
: When <rev> is not specified, the command annotates the changes starting backwards from the working tree copy. This flag makes the command pretend as if the working tree copy has the contents of the named file (specify `-` to make the command read from the standard input).

--date <format>
: The value is one of the following alternatives: {relative,local,default,iso,rfc,short}. 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 more information, See the discussion of the --date option at git-log(1).

-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.

-h
: Show help message.

-c
: Use the same output mode as 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 author name (Default: off).

-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 <n>+1 digits. Note that 1 column is used for a caret to mark the boundary commit.

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 is 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

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, it is used to map author and committer names and email addresses to canonical real names and email addresses.

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.

Example 1: 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 proper `.mailmap` file would look like:

```
Jane Doe         <jane@desktop.(none)>
Joe R. Developer <joe@example.com>
```

Note how there is no need for an entry for `<jane@laptop.(none)>`, because the real name of that author is already correct.

Example 2: Your repository contains commits from the following authors:

```
nick1 <bugs@company.xx>
nick2 <bugs@company.xx>
nick2 <nick2@company.xx>
santa <me@company.xx>
claus <me@company.xx>
CTO <cto@coompany.xx>
```

Then you might want a `.mailmap` file that looks like:

```
<cto@company.xx>                       <cto@coompany.xx>
Some Dude <some@dude.xx>         nick1 <bugs@company.xx>
Other Author <other@author.xx>   nick2 <bugs@company.xx>
Other Author <other@author.xx>         <nick2@company.xx>
Santa Claus <santa.claus@northpole.xx> <me@company.xx>
```

Use hash `#` for comments that are either on their own line, or after the email address.

SEE ALSO

git-annotate(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-branch(1) Manual Page

NAME

git-branch - List, create, or delete branches

SYNOPSIS

git branch [--color[=<when>] | --no-color] [-r | -a]
        [--list] [-v [--abbrev=<length> | --no-abbrev]]
        [--column[=<options>] | --no-column]
        [(--merged | --no-merged | --contains) [<commit>]] [<pattern>...]
git branch [--set-upstream | --track | --no-track] [-l] [-f] <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 (-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 with an asterisk. 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 is 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). 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.

Note that this will create the new branch, but it will not switch the working tree to it; use "git checkout <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.

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 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`.

-D
: Delete a branch irrespective of its merged status.

-l

--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.

-f

--force
: Reset <branchname> to <startpoint> if <branchname> exists already. Without `-f` *git branch* refuses to change an existing branch.

-m

--move
: Move/rename a branch and the corresponding reflog.

-M
: Move/rename a branch even if the new branch name already exists.

--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`.

--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.

-a

--all
: List both remote-tracking branches and local branches.

--list
: Activate the list mode. `git branch <pattern>` would try to create a branch, use `git branch --list <pattern>` to list matching branches.

-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 name of the upstream branch, as well (see also `git remote show <remote>`).

-q

--quiet
: Be more quiet when creating or deleting a branch, suppressing non-error messages.

--abbrev=<length>
: Alter the sha1's minimum display length in the output listing. 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
: When creating a new branch, set up `branch.<name>.remote` and `branch.<name>.merge` configuration entries to mark the start-point branch as "upstream" from 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.

  This behavior is the default when the start point is a remote-tracking branch. Set the branch.autoSetupMerge configuration variable to `false` if you want `git checkout` and `git branch` to always behave as if *--no-track* were given. Set it to `always` if you want this behavior when the start-point is either a local or remote-tracking branch.

--no-track
: Do not set up "upstream" configuration, even if the branch.autoSetupMerge configuration variable is true.

--set-upstream
: If specified branch does not exist yet or if `--force` has been given, acts exactly like `--track`. Otherwise sets up configuration like `--track` would when creating the branch, except that where branch points to is not changed.

-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. `request-pull`).

--contains [<commit>]
: Only list branches which 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 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 to rename.

<newbranch>
: The new name for an existing branch. The same restrictions as for <branchname> apply.

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   <1>
$ git checkout my2.6.14
```

1. 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   <1>
$ git branch -D test                                    <2>
```

1. 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 git-fetch(1).
2. 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.

Notes

If you are creating a branch that you want to checkout immediately, it is easier to use the git checkout command with its `-b` option to create a branch and check it out with a single command.

The options `--contains`, `--merged` and `--no-merged` serve three 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>.
- `--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.

SEE ALSO

git-check-ref-format(1), git-fetch(1), git-remote(1), "Understanding history: What is a branch?" in the Git User's Manual.

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-bundle(1) Manual Page

NAME

git-bundle - Move objects and refs by archive

SYNOPSIS

git bundle create <file> <git-rev-list-args>
git bundle verify <file>
git bundle list-heads <file> [<refname>...]
git bundle unbundle <file> [<refname>...]

DESCRIPTION

Some workflows require that one or more branches of development on one machine be replicated on another machine, but the two machines cannot be directly connected, and therefore the interactive Git protocols (git, ssh, rsync, http) cannot be used. This command provides support for *git fetch* and *git pull* to operate by packaging objects and references in an archive at the originating machine, then importing those into another repository using *git fetch* and *git pull* after moving the archive by some means (e.g., by sneakernet). As no direct connection between the repositories exists, the user must specify a basis for the bundle that is held by the destination repository: the bundle assumes that all objects in the basis are already in the destination repository.

OPTIONS

create <file>
: Used to create a bundle named *file*. This requires the *git-rev-list-args* arguments to define the bundle contents.

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. *git bundle* prints a list of missing commits, if any, and exits with a non-zero status.

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.

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*.

<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*).

SPECIFYING REFERENCES

git bundle will only package references that are shown by *git show-ref*: this includes heads, tags, and remote heads. References such as `master~1` cannot be packaged, but are perfectly suitable for defining the basis. More than one reference may be packaged, and more than one basis can be specified. The objects packaged are those not contained in the union of the given bases. Each basis can be specified explicitly (e.g. `^master~10`), or implicitly (e.g. `master~10..master`, `--since=10.days.ago master`).

It is very important that the basis used be held by the destination. 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.

EXAMPLE

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 basis. 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 basis, 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 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 basis 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 basis:

```
$ 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
```

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-cat-file(1) Manual Page

NAME

git-cat-file - Provide content or type and size information for repository objects

SYNOPSIS

> *git cat-file* (-t | -s | -e | -p | <type> | --textconv) <object>
> *git cat-file* (--batch | --batch-check) < <list-of-objects>

DESCRIPTION

In its first form, the command provides the content or the type of an object in the repository. The type is required unless *-t* or *-p* is used to find the object type, or *-s* is used to find the object size, or *--textconv* is used (which implies type "blob").

In the second form, a list of objects (separated by linefeeds) is provided on stdin, and the SHA-1, type, and size of each object is printed on stdout.

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 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>.

-e
: Suppress all output; instead exit with zero status if <object> exists and is a valid object.

-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.

--textconv
: Show the content as transformed by a textconv filter. In this case, <object> has be of the form <tree-ish>:<path>, or :<path> in order to apply the filter to the content recorded in the index at <path>.

--batch

--batch=<format>
: Print object information and contents for each object provided on stdin. May not be combined with any other options or arguments. See the section `BATCH OUTPUT` below for details.

--batch-check

--batch-check=<format>
: Print object information for each object provided on stdin. May not be combined with any other options or arguments. See the section `BATCH OUTPUT` below for details.

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.

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 git-rev-parse(1).

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 40-hex object name of the object.

`objecttype`
: The type of 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 40-hex sha1 of the delta base object. Otherwise, expands to the null sha1 (40 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, 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:

```
<sha1> SP <type> SP <size> LF
<contents> LF
```

Whereas `--batch-check='%(objectname) %(objecttype)'` would produce:

```
<sha1> 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
```

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 git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-check-attr(1) Manual Page

NAME

git-check-attr - Display gitattributes information

SYNOPSIS

> *git check-attr* [-a | --all | attr…] [--] pathname…
> *git check-attr* --stdin [-z] [-a | --all | attr…] < <list-of-paths>

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 file names from stdin instead of from the command-line.

-z
: The output format is modified to be machine-parseable. If `--stdin` is also given, input paths are separated with a NUL character instead of a linefeed character.

--
: 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 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

gitattributes(5).

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-check-ignore(1) Manual Page

NAME

git-check-ignore - Debug gitignore / exclude files

SYNOPSIS

> *git check-ignore* [options] pathname…
> *git check-ignore* [options] --stdin < <list-of-paths>

DESCRIPTION

For each pathname given via the command-line or from a file via `--stdin`, show the pattern from .gitignore (or other input files to the exclude mechanism) that decides if the pathname is excluded or included. Later patterns within a file take precedence over earlier ones.

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
: Also output details about the matching pattern (if any) for each given pathname.

--stdin
: Read file names from stdin instead of from the command-line.

-z
: The output format is modified to be machine-parseable (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 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

gitignore(5) gitconfig(5) git-ls-files(1)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-check-mailmap(1) Manual Page

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.

MAPPING AUTHORS

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, it is used to map author and committer names and email addresses to canonical real names and email addresses.

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.

Example 1: 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 proper `.mailmap` file would look like:

```
Jane Doe         <jane@desktop.(none)>
Joe R. Developer <joe@example.com>
```

Note how there is no need for an entry for `<jane@laptop.(none)>`, because the real name of that author is already correct.

Example 2: Your repository contains commits from the following authors:

```
nick1 <bugs@company.xx>
nick2 <bugs@company.xx>
nick2 <nick2@company.xx>
santa <me@company.xx>
claus <me@company.xx>
CTO <cto@coompany.xx>
```

Then you might want a `.mailmap` file that looks like:

```
<cto@company.xx>                       <cto@coompany.xx>
Some Dude <some@dude.xx>         nick1 <bugs@company.xx>
Other Author <other@author.xx>   nick2 <bugs@company.xx>
Other Author <other@author.xx>         <nick2@company.xx>
Santa Claus <santa.claus@northpole.xx> <me@company.xx>
```

Use hash `#` for comments that are either on their own line, or after the email address.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-checkout(1) Manual Page

NAME

git-checkout - Checkout a branch or paths to the working tree

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|--ours|--theirs|-m|--conflict=<style>] [<tree-ish>] [--] <paths>...
git checkout [-p|--patch] [<tree-ish>] [--] [<paths>...]

DESCRIPTION

Updates files in the working tree to match the version in the index or the specified tree. If no paths are 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, 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 a rather expensive side-effects to show only the tracking information, if exists, for the current branch.

git checkout -b|-B <new_branch> [<start point>]
: Specifying `-b` causes a new branch to be created as if 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.

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 [-p|--patch] [<tree-ish>] [--] <pathspec>...
: When <paths> or `--patch` are given, *git checkout* does **not** switch branches. It updates the named paths in the working tree from the index file or from a named <tree-ish> (most often a commit). In this case, the `-b` and `--track` options are meaningless and giving either of them results in an error. The <tree-ish> argument can be used to specify a specific tree-ish (i.e. commit, tag or tree) to update the index for the given paths before updating the working tree.

  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.

OPTIONS

-q

--quiet
: Quiet, suppress feedback messages.

-f

--force
: When switching branches, proceed even if the index or the working tree differs from HEAD. This is used to throw away local changes.

  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.

-b <new_branch>
: Create a new branch named <new_branch> and start it at <start_point>; see git-branch(1) for details.

-B <new_branch>
: Creates the branch <new_branch> and start it at <start_point>; if it already exists, then reset it to <start_point>. This is equivalent to running "git branch" with "-f"; see git-branch(1) for details.

-t

--track
: When creating a new branch, set up "upstream" configuration. See "--track" in 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.

-l
: Create the new branch's reflog; see git-branch(1) for details.

--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 *orphan* 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.

--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) and "diff3" (in addition to what is shown by "merge" style, shows the original contents).

-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 git-add(1) to learn how to operate the `--patch` mode.

<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).

  As a special case, the `"@{-N}"` syntax for the N-th last branch/commit checks out branches (instead of detaching). You may also specify `-` which is synonymous with `"@{-1}"`.

  As a further 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 git-branch(1) for details. Defaults to HEAD.

<tree-ish>

Tree to checkout from (when paths are given). If not specified, the index will be used.

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   <1>
$ git branch foo        <2>
$ git tag foo           <3>
```

1. 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.
2. similarly creates a new branch *foo*, which refers to commit *f*, but leaves HEAD detached.
3. 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
```

EXAMPLES

1. 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             <1>
   $ git checkout master~2 Makefile  <2>
   $ rm -f hello.c
   $ git checkout hello.c            <3>
   ```

   1. switch branch
   2. take a file out of another commit
   3. 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. 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. 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
```

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-checkout-index(1) Manual Page

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]
>   [-z] [--stdin]
>   [--] [<file>...]

DESCRIPTION

Will copy all files listed 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. 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 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.

--stdin
: Instead of taking list of paths from the command line, read 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 git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-check-ref-format(1) Manual Page

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 avoids ambiguities in certain reference name expressions (see 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, it expands the “previous branch syntax” `@{-n}`. For example, `@{-1}` is a way to refer the last branch you were on. 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.

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 place of a one full pathname component (e.g., `foo/*/bar` but not `foo/bar*`).

--normalize
: Normalize *refname* by removing any leading slash (`/`) characters and collapsing runs of adjacent slashes between name components into a single slash. Iff the normalized refname is valid then print it to standard output and exit with a status of 0. (`--print` is a deprecated way to spell `--normalize`.)

EXAMPLES

- Print the name of the previous branch:

```
$ 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") ||
die "we do not like '$newbranch' as a branch name."
```

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-cherry(1) Manual Page

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 git-cherry-pick(1), git-am(1) or 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

Patch workflows

git-cherry is frequently used in patch-based workflows (see 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
```

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`.

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

git-patch-id(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-cherry-pick(1) Manual Page

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[<key-id>]] <commit>...
> *git cherry-pick* --continue
> *git cherry-pick* --quit
> *git cherry-pick* --abort

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 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 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 gitrevisions(7). Sets of commits can be passed but no traversal is done by default, as if the *--no-walk* option was specified, see 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.

-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 Signed-off-by line at the end of the commit message.

-S[<key-id>]

--gpg-sign[=<key-id>]
: GPG-sign commits.

--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 are dropped. To force the inclusion of those commits use `--keep-redundant-commits`.

--allow-empty-message
: By default, cherry-picking a commit with an empty message will fail. This option overrides that behaviour, allowing commits with empty messages to be cherry picked.

--keep-redundant-commits
: If a commit being cherry picked duplicates a commit already in the current history, it will become empty. By default these redundant commits cause `cherry-pick` to stop so the user can examine the commit. This option overrides that behavior and creates an empty commit object. Implies `--allow-empty`.

--strategy=<strategy>
: Use the given merge strategy. Should only be used once. See the MERGE STRATEGIES section in git-merge(1) for details.

-X<option>

--strategy-option=<option>
: Pass the merge strategy-specific option through to the merge strategy. See git-merge(1) for details.

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.

--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^             <1>
$ git diff                           <2>
$ git reset --merge ORIG_HEAD        <3>
$ git cherry-pick -Xpatience topic^  <4>
```

1. 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.
2. summarize changes to be reconciled
3. cancel the cherry-pick. In other words, return to the pre-cherry-pick state, preserving any local modifications you had in the working tree.
4. try to apply the change introduced by `topic^` again, spending extra time to avoid mistakes based on incorrectly matching context lines.

SEE ALSO

git-revert(1)

GIT

Part of the git(1) suite

Last updated 2015-05-03 21:16:24 CEST

git-citool(1) Manual Page

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 git-gui(1) for more details.

GIT

Part of the git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-clean(1) Manual Page

NAME

git-clean - Remove untracked files from the working tree

SYNOPSIS

> *git clean* [-d] [-f] [-i] [-n] [-q] [-e <pattern>] [-x | -X] [--] <path>…

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 `<path>...` arguments are given, only those paths are affected.

OPTIONS

-d
: Remove untracked directories in addition to untracked files. If an untracked directory is managed by a different Git repository, it is not removed by default. Use -f option twice if you really want to remove such a directory.

-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, -n or -i. Git will refuse to delete directories with .git sub directory or file unless a second -f is given. This affects also git submodules where the storage area of the removed submodule under .git/modules/ is not removed until -f is given twice.

-i

--interactive
: Show what would be done and clean files interactively. See “Interactive mode” for details.

-n

--dry-run
: Don’t actually remove anything, just show what would be done.

-q

--quiet
: Be quiet, only report errors, but not the files that are successfully removed.

-e <pattern>

--exclude=<pattern>
: In addition to those found in .gitignore (per directory) and $GIT_DIR/info/exclude, also consider these patterns to be in the set of the ignore rules in effect.

-x
: Don’t use the standard ignore rules read from .gitignore (per directory) and $GIT_DIR/info/exclude, but do still use the ignore rules given with `-e` options. This allows removing all untracked files, including build products. This can be used (possibly in conjunction with *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 excludes files end 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 do cleaning.

help
: Show brief usage of interactive git-clean.

SEE ALSO

gitignore(5)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-clone(1) Manual Page

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]
    [--recursive | --recurse-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 -r`), 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

--local

-l

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.

--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.

--shared

-s

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 gc --auto`. (See git-gc(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 `-l` option in a repository cloned with `-s` will copy objects from the source repository into a pack in the cloned repository, removing the disk space savings of `clone -s`. It is safe, however, to run `git gc`, which uses the `-l` option by default.

If you want to break the dependency of a repository cloned with `-s` 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 <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.

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.

--quiet

-q

Operate quietly. Progress is not reported to the standard error stream. This flag is also passed to the 'rsync' command when given.

--verbose

-v

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 -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

--no-checkout

-n
: No checkout of HEAD is performed after the clone is complete.

--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 `-n` 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.

--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.

--origin <name>

-o <name>
: Instead of using the remote name `origin` to keep track of the upstream repository, use `<name>`.

--branch <name>

-b <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.

--upload-pack <upload-pack>

-u <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 git-init(1).)

--config <key>=<value>

-c <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 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.

--depth <depth>
: Create a *shallow* clone with a history truncated to the specified number of revisions.

--[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. When creating a shallow clone with the `--depth` option, this is the default, unless `--no-single-branch` is given to fetch the histories near the tips of all branches. 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.

--recursive

--recurse-submodules
: After the clone is created, initialize all submodules within, using their default settings. This is equivalent to running `git submodule update --init --recursive` 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)

--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.

<repository>
: The (possibly remote) repository to clone from. See the 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.

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 and rsync can be used for fetching and pushing, but these are 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.xz[:port]/path/to/repo.git/
- git://host.xz[:port]/path/to/repo.git/
- http[s]://host.xz[:port]/path/to/repo.git/
- ftp[s]://host.xz[:port]/path/to/repo.git/
- rsync://host.xz/path/to/repo.git/

An alternative scp-like syntax may also be used with the ssh protocol:

- [user@]host.xz:path/to/repo.git/

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.xz[:port]/~[user]/path/to/repo.git/
- git://host.xz[:port]/~[user]/path/to/repo.git/
- [user@]host.xz:/~[user]/path/to/repo.git/

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.

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 gitremote-helpers(1) 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
```

GIT

Part of the git(1) suite

Last updated 2015-01-25 20:40:15 CET

git-column(1) Manual Page

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 its input into multiple 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.

--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=<N>
: 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.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-commit(1) Manual Page

NAME

git-commit - Record changes to the repository

SYNOPSIS

```
git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
          [--dry-run] [(-c | -C | --fixup | --squash) <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] [-S[<key-id>]] [--] [<file>...]
```

DESCRIPTION

Stores the current contents of the index in a new commit along with a log message from the user describing the changes.

The content to be added can be specified in several ways:

1. by using *git add* to incrementally "add" changes to the index before using the *commit* command (Note: even modified files must be "added");
2. by using *git rm* 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, 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, before finalizing the operation. See the “Interactive Mode” section of 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 chose which changes to commit. See 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=<commit>
: Construct a commit message for use with `rebase --autosquash`. The commit message will be the subject line from the specified commit with a prefix of "fixup! ". See 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 git-rebase(1) for details.

--reset-author
: When used with -C/-c/--amend options, or when committing after a a conflicting cherry-pick, declare that the authorship of the resulting commit now belongs of the committer. This also renews the author timestamp.

--short
: When doing a dry-run, give the output in the short-format. See 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 git-status(1) for details. Implies `--dry-run`.

--long
: When doing a dry-run, give the output in a the long-format. Implies `--dry-run`.

-z

--null
: When showing `short` or `porcelain` status output, terminate entries in the status output with NUL, instead of LF. If no format is given, implies the `--porcelain` output format.

-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.

-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
: Add Signed-off-by line by the committer at the end of the commit log message.

-n

--no-verify
: This option bypasses the pre-commit and commit-msg hooks. See also 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 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, and #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 "`# ------------------------ >8 ------------------------`" is truncated if the message is to be edited. "`#`" can be customized with core.commentChar.

  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 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 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.

-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.

    The default can be changed using the status.showUntrackedFiles configuration variable documented in 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. Note that this diff output doesn't have its lines prefixed with *#*.

  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 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 git-status(1) in the commit message template when using an editor to prepare the default commit message.

-S[<keyid>]

--gpg-sign[=<keyid>]
: GPG-sign commit.

--no-gpg-sign
: Countermand `commit.gpgSign` configuration variable that is set to force each and every commit to be signed.

--
: Do not interpret any more arguments as options.

<file>...
: When files are given on the command line, the command commits the contents of the named files, without recording the changes already staged. The contents of these files are also staged for the next commit on top of what have been staged before.

DATE FORMATS

The GIT_AUTHOR_DATE, GIT_COMMITTER_DATE environment variables and the `--date` option 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 2 hours ahead UTC) is `+0200`.

RFC 2822
: The standard email 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.

Note In addition, the date part is accepted in the following formats: `YYYY.MM.DD`, `MM/DD/YYYY` and `DD.MM.YYYY`.

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 reset HEAD -- <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).

DISCUSSION

Though not required, it's a good idea to begin the commit message with a single short (less than 50 character) 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, 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.

At the core level, Git is character encoding agnostic.

- The pathnames recorded in the index and in the tree objects are treated as uninterpreted sequences of non-NUL bytes. What readdir(2) returns are what are recorded and compared with the data Git keeps track of, which in turn are expected to be what lstat(2) and creat(2) accepts. There is no such thing as pathname encoding translation.
- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- The commit log messages are uninterpreted sequences of non-NUL bytes.

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* issues 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 its `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 git-var(1) for details.

HOOKS

This command can run `commit-msg`, `prepare-commit-msg`, `pre-commit`, and `post-commit` hooks. See 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

git-add(1), git-rm(1), git-mv(1), git-merge(1), git-commit-tree(1)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-commit-tree(1) Manual Page

NAME

git-commit-tree - Create a new commit object

SYNOPSIS

```
git commit-tree <tree> [(-p <parent>)...] < changelog
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 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.

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.

-S[<keyid>]

--gpg-sign[=<keyid>]
: GPG-sign commit.

--no-gpg-sign
: Countermand `commit.gpgSign` configuration variable that is set to force each and every commit to be signed.

Commit Information

A commit encapsulates:

- all parent object ids
- author name, email and date
- committer name and email and the commit time.

While parent object ids are provided on the command line, 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)

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).

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, 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 2 hours ahead UTC) is `+0200`.

RFC 2822
: The standard email 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.

Note In addition, the date part is accepted in the following formats: `YYYY.MM.DD`, `MM/DD/YYYY` and `DD.MM.YYYY`.

Discussion

At the core level, Git is character encoding agnostic.

- The pathnames recorded in the index and in the tree objects are treated as uninterpreted sequences of non-NUL bytes. What readdir(2) returns are what are recorded and compared with the data Git keeps track of, which in turn are expected to be what lstat(2) and creat(2) accepts. There is no such thing as pathname encoding translation.
- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- The commit log messages are uninterpreted sequences of non-NUL bytes.

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* issues 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 its `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

git-write-tree(1)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-config(1) Manual Page

NAME

git-config - Get and set repository or global options

SYNOPSIS

git config [<file-option>] [type] [-z|--null] name [value [value_regex]]
git config [<file-option>] [type] --add name value
git config [<file-option>] [type] --replace-all name value [value_regex]
git config [<file-option>] [type] [-z|--null] --get name [value_regex]
git config [<file-option>] [type] [-z|--null] --get-all name [value_regex]
git config [<file-option>] [type] [-z|--null] --get-regexp name_regex [value_regex]
git config [<file-option>] [type] [-z|--null] --get-urlmatch name URL
git config [<file-option>] --unset name [value_regex]
git config [<file-option>] --unset-all name [value_regex]
git config [<file-option>] --rename-section old_name new_name
git config [<file-option>] --remove-section name
git config [<file-option>] [-z|--null] -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 POSIX regexp `value_regex` needs to be given. Only the existing values that match the regexp are updated or unset. If you want to handle the lines that do **not** match the regex, just prepend a single exclamation mark in front (see also [EXAMPLES]).

The type specifier can be either *--int* or *--bool*, to make *git config* ensure that the variable(s) are of the given type and convert the value to the canonical form (simple decimal number for int, a "true" or "false" string for bool), or *--path*, which does some path expansion (see *--path* below). If no type specifier is passed, no checks or transformations are performed on the value.

When reading, the values are read from the system, global and repository local configuration files by default, and options *--system*, *--global*, *--local* and *--file <filename>* can be used to tell the command to read from only that location (see [FILES]).

When writing, the new value is written to the repository local configuration file by default, and options *--system*, *--global*, *--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:

1. The config file is invalid (ret=3),
2. can not write to the config file (ret=4),
3. no section or name was provided (ret=2),
4. the section or key is invalid (ret=1),
5. you try to unset an option which does not exist (ret=5),
6. you try to unset/set an option for which multiple lines match (ret=5), or
7. you try to use an invalid regexp (ret=6).

On success, the command returns the exit code 0.

OPTIONS

--replace-all
: Default behavior is to replace at most one line. This replaces all lines matching the key (and optionally the value_regex).

--add
: Adds a new line to the option without altering any existing values. This is the same as providing *^$* as the value_regex in `--replace-all`.

--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 does not fail if the number of values for the key is not exactly one.

--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 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.

--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 [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 [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 [FILES].

-f config-file

--file config-file
: Use the given config file instead of the one specified by GIT_CONFIG.

--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 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.

--bool
: *git config* will ensure that the output is "true" or "false"

--int
: *git config* will ensure that the output is a simple decimal number. An optional value suffix of *k*, *m*, or *g* in the config file will cause the value to be multiplied by 1024, 1048576, or 1073741824 prior to output.

--bool-or-int
: *git config* will ensure that the output matches the format of either --bool or --int, as described above.

--path
: *git-config* will expand leading *~* to the value of *$HOME*, and *~user* to the home directory for the specified user. This option has no effect when setting the value (but you can use *git config bla ~/* from the command line to let your shell do the expansion).

-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.

--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`.

-e

--edit
: Opens an editor to modify the specified config file; either *--system*, *--global*, or repository (default).

--[no-]includes
: Respect `include.*` directives in config files when looking up values. Defaults to on.

FILES

If not set explicitly with *--file*, there are four files where *git config* will search for configuration options:

$(prefix)/etc/gitconfig
: System-wide configuration file.

$XDG_CONFIG_HOME/git/config
: Second user-specific configuration file. If $XDG_CONFIG_HOME is not set or empty, `$HOME/.config/git/config` will be used. Any single-valued variable set in this file will be overwritten by whatever is in `~/.gitconfig`. It is a good idea not to create this file if you sometimes use older versions of Git, as support for this file was added fairly recently.

~/.gitconfig
: User-specific configuration file. Also called "global" configuration file.

$GIT_DIR/config

Repository specific configuration file.

If no further options are given, all reading options will read all of these files that are available. If the global or the system-wide configuration file are not available they will be ignored. If the repository configuration file is not available or readable, *git config* will exit with a non-zero error code. However, in neither case will an error message be issued.

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.

All writing options will per default write 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 override these rules either by command-line options or by environment variables. The *--global* and the *--system* options will limit the file used to the global or system-wide file respectively. The GIT_CONFIG environment variable has a similar effect, but you can specify any filename you want.

ENVIRONMENT

GIT_CONFIG
: Take the configuration from the given file instead of .git/config. Using the "--global" option forces this to ~/.gitconfig. Using the "--system" option forces this to $(prefix)/etc/gitconfig.

GIT_CONFIG_NOSYSTEM
: Whether to skip reading settings from the system-wide $(prefix)/etc/gitconfig file. See git(1) for details.

See also [FILES].

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 --bool --get-urlmatch http.sslverify https://good.example.com
true
% git config --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 `.git/config` file in each repository is 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 porcelains. 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.

Syntax

The syntax is fairly flexible and permissive; whitespaces 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 (doublequote `"` and backslash can be included by escaping them as `\"` and `\\`, respectively). 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.

A line that defines a value can be continued to the next line by ending it with a `\`; the backquote and the end-of-line are stripped. Leading whitespaces after *name =*, the remainder of the line after the first comment character *#* or *;*, and trailing whitespaces of the line are discarded unless they are enclosed in double quotes. Internal whitespaces within the value are retained verbatim.

Inside double quotes, 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.

Includes

You can include one config file from another by setting the special `include.path` variable to the name of the file to be included. The included file is expanded immediately, as if its contents had been found at the location of the include directive. If the value of the `include.path` variable is a relative path, the path is considered to be relative to the configuration file in which the include directive was found. The value of `include.path` is subject to tilde expansion: `~/` is expanded to the value of `$HOME`, and `~user/` to the specified user's home directory. See below for examples.

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 ; expand "foo" relative to the current file
        path = ~/foo ; expand "foo" in your $HOME directory
```

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 can be spelled as `yes`, `on`, `true`, or `1`. Also, a variable defined without `= <value>` is taken as true.

  false
  : Boolean false can be spelled as `no`, `off`, `false`, or `0`.

    When converting value to the canonical form using *--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 variables that takes a color is a list of colors (at most two) and attributes (at most one), separated by spaces. The colors accepted are `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` and `white`; the attributes are `bold`, `dim`, `ul`, `blink` and `reverse`. The first color given is the foreground; the second is the background. The position of the attribute, if any, doesn't matter. Attributes may be turned off specifically by prefixing them with `no` (e.g., `noreverse`, `noul`, etc).

  Colors (foreground and background) 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 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.

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.

advice.*
: These variables control various optional help messages designed to aid new users. All *advice.** variables default to *true*, and you can tell Git that you do not need help by setting these to *false*:

  pushUpdateRejected
  : Set this variable to *false* if you want to disable *pushNonFFCurrent*, *pushNonFFMatching*, *pushAlreadyExists*, *pushFetchFirst*, and *pushNeedsForce* simultaneously.

  pushNonFFCurrent
  : Advice shown when git-push(1) fails due to a non-fast-forward update to the current branch.

  pushNonFFMatching
  : Advice shown when you ran git-push(1) and pushed *matching refs* explicitly (i.e. you used *:*, or specified a refspec that isn't your current branch) and it resulted in a non-fast-forward error.

  pushAlreadyExists
  : Shown when git-push(1) rejects an update that does not qualify for fast-forwarding (e.g., a tag.)

  pushFetchFirst
  : Shown when 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 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.

  statusHints
  : Show directions on how to proceed from the current state in the output of git-status(1), in the template shown when writing commit messages in git-commit(1), and in the help message shown by git-checkout(1) when switching branch.

  statusUoption
  : Advise to consider using the `-u` option to git-status(1) when the command takes more than 2 seconds to enumerate untracked files.

  commitBeforeMerge
  : Advice shown when git-merge(1) refuses to merge to avoid overwriting local changes.

resolveConflict
: Advice shown by various commands when conflicts prevent the operation from being performed.

implicitIdentity
: Advice on how to set your identity configuration when your information is guessed from the system username and domain name.

detachedHead
: Advice shown when you used git-checkout(1) to move to the detach HEAD state, to instruct how to create a local branch after the fact.

amWorkDir
: Advice that shows the location of the patch file when git-am(1) fails to apply it.

rmHints
: In case of failure in the output of git-rm(1), show directions on how to proceed from the current state.

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 an non-executable file with executable bit on. git-clone(1) or 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 git-update-index(1).

  The default is true (when core.filemode is not specified in the config file).

core.ignoreCase
: If true, this option enables various workarounds to enable Git to work better on filesystems that are not case sensitive, like FAT. 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 git-clone(1) or git-init(1) will probe and set core.ignoreCase true if appropriate when the repository is created.

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.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 git-update-index(1). True by default.

core.checkStat
: Determines which stat fields to match between the index and work tree. The user can set this to *default* or *minimal*. Default (or explicitly *default*), is to check all fields, including the sub-second part of mtime and ctime.

core.quotePath
: The commands that output paths (e.g. *ls-files*, *diff*), when not given the `-z` option, will quote "unusual" characters in the pathname by enclosing the pathname in a double-quote pair and with backslashes the same way strings in C source code are quoted. If this variable is set to false, the bytes higher than 0x80 are not quoted but output as verbatim. Note that double quote, backslash and control characters are always quoted without `-z` regardless of the setting of this variable.

core.eol
: Sets the line ending type to use in the working directory for files that have the `text` property set. Alternatives are *lf*, *crlf* and *native*, which uses the platform's native line ending. The default value is `native`. See gitattributes(5) for more information on end-of-line conversion.

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 almost the same as setting the `text` attribute to "auto" on all files except that text files are not guaranteed to be normalized: files that contain `CRLF` in the repository will not be touched. Use this setting if you want to have `CRLF` line endings in your working directory even though the repository does not have normalized line endings. This variable can be set to *input*, in which case no output conversion is performed.

core.symlinks
: If false, symbolic links are checked out as small plain files that contain the link text. git-update-index(1) and 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 git-clone(1) or 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.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 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.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 git-add(1) or git-merge(1).

  This setting is automatically guessed by git-clone(1) or 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. 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.

  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 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 8 GiB 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 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
: Files larger than this size are stored deflated, without attempting delta compression. Storing large files without delta compression avoids excessive memory usage, at the slight expense of increased disk usage. Additionally files larger than this size are always treated as binary.

  Default is 512 MiB on all platforms. This should be reasonable for most projects as source code and other text files can still be delta compressed, but larger binary media files won't be.

  Common unit suffixes of *k*, *m*, or *g* are supported.

core.excludesFile
: In addition to *.gitignore* (per-directory) and *.git/info/exclude*, Git looks into this file for patterns of files which are not meant to be tracked. "`~/`" is expanded to the value of `$HOME` and "`~user/`" to the specified user's home directory. 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. See 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 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.editor
: Commands such as `commit` and `tag` that lets you edit messages by launching an editor uses the value of this variable when it is set, and the environment variable `GIT_EDITOR` is not set. See git-var(1).

core.commentChar
: Commands such as `commit` and `tag` that lets 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.

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.

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.fsyncObjectFiles
: This boolean will enable *fsync()* when writing object files.

  This is a total waste of time and effort on a filesystem that orders data writes properly, but can be useful for filesystems that do not use journalling (traditional UNIX filesystems) or that only journal metadata and not file contents (OS X's HFS+, or Linux ext3 with "data=writeback").

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.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 git-notes(1).

core.sparseCheckout
: Enable "sparse checkout" feature. See section "Sparse checkout" in git-read-tree(1) for more information.

core.abbrev
: Set the length object names are abbreviated to. If unspecified, many commands abbreviate to 7 hexdigits, which may not be enough for abbreviated object names to stay unique for sufficiently long time.

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 git-add(1). `add.ignore-errors` is deprecated, as it does not follow the usual naming convention for configuration variables.

alias.*
: Command aliases for the 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 is supported. A quote pair or a backslash can be used to quote them.

  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 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 git-am(1), git-mailsplit(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 tells *git apply* to respect all whitespace differences. See git-apply(1).

apply.whitespace
: Tells *git apply* how to handle whitespaces, in the same way as the *--whitespace* option. See git-apply(1).

branch.autoSetupMerge
: Tells *git branch* and *git checkout* to set up new branches so that 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. This option defaults to true.

branch.autoSetupRebase
: When a new branch is created with *git branch* 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.<name>.remote
: When on branch <name>, it tells *git fetch* and *git push* which remote to fetch from/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, 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 at 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 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 preserve, also pass `--preserve-merges` along to 'git rebase'
so that locally committed merge commits will not be flattened
by running 'git pull'.
```

NOTE: this is a possibly dangerous operation; do **not** use it unless you understand the implications (see git-rebase(1) for details).

branch.<name>.description
: Branch description, can be edited with `git branch --edit-description`. Branch description is automatically added in 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 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 git-help(1)) or a working repository in gitweb (see git-instaweb(1)).

clean.requireForce
: A boolean to make git-clean do nothing unless given -f, -i or -n. Defaults to true.

color.branch
: A boolean to enable/disable color in the output of 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. Defaults to false.

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`, git-diff(1), git-log(1), and 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. Defaults to false.

  This does not affect 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 `plain` (context text), `meta` (metainformation), `frag` (hunk header), *func* (function in hunk header), `old` (removed lines), `new` (added lines), `commit` (commit headers), or `whitespace` (highlighting whitespace errors).

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.

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. Defaults to `false`.

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`)

  `match`
  : matching text (same as setting `matchContext` and `matchSelected`)

  `matchContext`
  : matching text in context lines

  `matchSelected`
  : matching text in selected lines

  `selected`
  : non-matching text in selected lines

  `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. Defaults to false.

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 enable/disable colored output when the pager is in use (default is true).

color.showBranch
: A boolean to enable/disable color in the output of 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. Defaults to false.

color.status
: A boolean to enable/disable color in the output of 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. Defaults to false.

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), or `unmerged` (files which have unmerged changes).

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 list 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 listing 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 git-commit(1) for details. Changing the default can be useful when you always want to keep lines that begin with 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 a file to use as the template for new commit messages. "`~/`" is expanded to the value of `$HOME` and "`~user/`" to the specified user's home directory.

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. See gitcredentials(7) for details.

credential.useHttpPath
: When acquiring credentials, consider the "path" component of an http or https URL to be important. Defaults to false. See 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 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 gitcredentials(7) for details on how URLs are matched.

diff.autoRefreshIndex
: When using *git diff* to compare with work tree files, do not consider stat-only change 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 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.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.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 git(1). Note: if you want to use an external diff program only on a subset of your files, you might want to use 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* also honors 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.

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.orderFile
: File indicating how to order files within a diff, using one shell glob pattern per line. Can be overridden by the *-O* option to git-diff(1).

diff.renameLimit
: The number of files to consider when performing the copy/rename detection; equivalent to the *git diff* option *-l*.

diff.renames
: Tells Git to detect renames. If set to any boolean value, it will enable basic rename detection. If set to "copies" or "copy", it will detect copies, as well.

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 "log" format lists the commits in the range like git-submodule(1) `summary` does. The "short" format format just shows the names of the commits at the beginning and end of the range. 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 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 gitattributes(5) for details.

diff.<driver>.binary
: Set this option to true to make the diff driver treat files as binary. See 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 gitattributes(5) for details.

diff.<driver>.wordRegex
: The regular expression that the diff driver should use to split words in a line. See gitattributes(5) for details.

diff.<driver>.cachetextconv
: Set this option to true to make the diff driver cache the text conversion outputs. See gitattributes(5) for details.

diff.tool
: Controls which diff tool is used by 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.

- araxis
- bc
- bc3
- codecompare
- deltawalker
- diffmerge
- diffuse
- ecmerge
- emerge
- gvimdiff
- gvimdiff2
- gvimdiff3
- kdiff3
- kompare
- meld
- opendiff
- p4merge
- tkdiff
- vimdiff
- vimdiff2
- vimdiff3
- xxdiff

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".

difftool.<tool>.path
: Override the path for the given tool. This is useful in case your tool is not in the PATH.

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.

difftool.prompt
: Prompt before each invocation of the diff tool.

fetch.recurseSubmodules
: This option can be either set to a boolean value or to *on-demand*. Setting it to a boolean changes the behavior of fetch and pull to unconditionally recurse into submodules when set to true or to not recurse at all when set to false. When set to *on-demand* (the default value), fetch and pull will only recurse into a populated submodule when its superproject retrieves a commit that updates the submodule's reference.

fetch.fsckObjects
: If it is set to true, git-fetch-pack will check all fetched objects. It will abort in the case of a malformed object or a broken link. The result of an abort are only dangling objects. Defaults to false. If not set, the value of `transfer.fsckObjects` is used instead.

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`.

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 git-format-patch(1).

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 git-format-patch(1).

format.headers
: Additional email headers to include in a patch to be submitted by mail. See 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 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.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.pretty
: The default pretty format for log/show/whatchanged command, See git-log(1), git-show(1), 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: line 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.

filter.<driver>.clean
: The command which is used to convert the content of a worktree file to a blob upon checkin. See 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 gitattributes(5) for details.

gc.aggressiveDepth
: The depth parameter used in the delta compression algorithm used by *git gc --aggressive*. This defaults to 250.

gc.aggressiveWindow
: The window size parameter used in the delta compression algorithm used by *git gc --aggressive*. This defaults to 250.

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 it.

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.

gc.autoDetach
: Make `git gc --auto` return immediately and run in background if the system supports it. Default is true.

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.pruneExpire
: When *git gc* is run, it will call *prune --expire 2.weeks.ago*. 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.

gc.reflogExpire

gc.<pattern>.reflogExpire
: *git reflog expire* removes reflog entries older than this time; defaults to 90 days. With "<pattern>" (e.g. "refs/stash") in the middle the setting applies only to the refs that match the <pattern>.

gc.reflogExpireUnreachable

gc.<ref>.reflogExpireUnreachable
: *git reflog expire* removes reflog entries older than this time and are not reachable from the current tip; defaults to 30 days. With "<pattern>" (e.g. "refs/stash") in the middle, the setting applies only to the refs that match the <pattern>.

gc.rerereResolved
: Records of conflicted merge you resolved earlier are kept for this many days when *git rerere gc* is run. The default is 60 days. See 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. The default is 15 days. See 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 git-cvsserver(1).

gitcvs.logFile
: Path to a log file where the CVS server interface well... logs various stuff. See 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 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 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 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 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 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 gitweb(1) for description.

gitweb.avatar

gitweb.blame

gitweb.grep

gitweb.highlight

gitweb.patches

gitweb.pickaxe

gitweb.remote_heads

gitweb.showSizes

gitweb.snapshot
: See gitweb.conf(5) for description.

grep.lineNumber
: If set to true, enable *-n* 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 return to the default matching behavior.

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*.

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 $file - <$signature" is run, and the program is expected to signal a good signature by exiting with code 0, and 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.

gui.commitMsgWidth
: Defines how wide the commit message window is in the git-gui(1). "75" is the default.

gui.diffContext
: Specifies how many context lines should be used in calls to diff made by the git-gui(1). The default is "5".

gui.displayUntracked
: Determines if :git-gui(1) shows untracked files in the file list. The default is "true".

gui.encoding
: Specifies the default encoding to use for displaying of file contents in git-gui(1) and gitk(1). It can be overridden by setting the *encoding* attribute for relevant files (see gitattributes(5)). If this option is not set, the tools default

to the locale encoding.

gui.matchTrackingBranch
: Determines if new branches created with git-gui(1) should default to tracking remote branches with matching names or not. Default: "false".

gui.newBranchTemplate
: Is used as suggested name when creating new branches using the git-gui(1).

gui.pruneDuringFetch
: "true" if git-gui(1) should prune remote-tracking branches when performing a fetch. The default value is "false".

gui.trustmtime
: Determines if 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 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 git-blame(1) manual for more information on copy detection.

gui.blamehistoryctx
: Specifies the radius of history context in days to show in 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 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 git-help(1).

help.format
: Override the default help format used by git-help(1). Values *man*, *info*, *web* and *html* are supported. *man* is the default. *web* and *html* are the same.

help.autoCorrect
: Automatically correct and execute mistyped commands after waiting for the given number of deciseconds (0.1 sec). If more than one command can be deduced from the entered text, nothing will be executed. If the value of this option is negative, the corrected command will be executed immediately. If the value is 0 - the command will be just shown but not executed. This is the default.

http.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)`). This can be overridden on a per-remote basis; see remote.<name>.proxy

http.cookieFile
: 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 only used 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.sslVerify
: Whether to verify the SSL certificate when fetching or pushing over HTTPS. 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.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.

http.lowSpeedLimit, http.lowSpeedTime
: If the HTTP transfer speed 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 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 client Git 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.<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 exactly between the config key and the URL.
  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 at other places in the future or in other porcelains). See e.g. git-mailinfo(1). Defaults to *utf-8*.

i18n.logOutputEncoding
: Character encoding the commit messages are converted to when running *git log* and friends.

imap
: The configuration variables in the *imap* section are described in git-imap-send(1).

index.version
: Specify the version with which new index files should be initialized. This does not affect existing repositories.

init.templateDir
: Specify the directory from which templates will be copied. (See the "TEMPLATE DIRECTORY" section of git-init(1).)

instaweb.browser
: Specify the program that will be used to browse your working repository in gitweb. See git-instaweb(1).

instaweb.httpd
: The HTTP daemon command-line to start gitweb on your working repository. See git-instaweb(1).

instaweb.local
: If true the web server started by git-instaweb(1) will be bound to the local IP (127.0.0.1).

instaweb.modulePath
: The default module path for 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 git-instaweb(1).

interactive.singleKey
: In interactive commands, allow the user to provide one-letter input with a single key (i.e., without hitting enter). Currently this is used by the `--patch` mode of git-add(1), git-checkout(1), git-commit(1), git-reset(1), and git-stash(1). Note that this setting is silently ignored if portable keystroke input is not available; requires the Perl module Term::ReadKey.

log.abbrevCommit
: If true, makes git-log(1), git-show(1), and 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. Possible values are `relative`, `local`, `default`, `iso`, `rfc`, and `short`; see git-log(1) for details.

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. This is the same as the log commands *--decorate* option.

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 git-log(1) or git-whatchanged(1), which normally hide the root commit will now show it. True by default.

log.mailmap
: If true, makes git-log(1), git-show(1), and git-whatchanged(1) assume `--use-mailmap`.

mailinfo.scissors
: If true, makes git-mailinfo(1) (and therefore git-am(1)) act by default as if the --scissors option was provided on the command-line. When active, this features 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 git-shortlog(1) and 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.

man.viewer
: Specify the programs that may be used to display help in the *man* format. See 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 argument. (See 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 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.

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.

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.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.renameLimit
: The number of files to consider when performing rename detection during a merge; if not specified, defaults to the value of diff.renameLimit.

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 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.tool
: Controls which merge tool is used by 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.

- araxis
- bc
- bc3
- codecompare
- deltawalker
- diffmerge
- diffuse
- ecmerge
- emerge
- gvimdiff
- gvimdiff2
- gvimdiff3
- kdiff3
- meld
- opendiff
- p4merge
- tkdiff
- tortoisemerge
- vimdiff
- vimdiff2
- vimdiff3
- xxdiff

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 gitattributes(5) for details.

merge.<driver>.driver
: Defines the command that implements a custom low-level merge driver. See 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 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>.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 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.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.

notes.displayRef
: The (fully qualified) refname from which to show notes when showing commit messages. The value of this variable can be set to a glob, in which case notes from all matching refs will be shown. You may also specify this configuration variable several times. 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 overridden with the `GIT_NOTES_DISPLAY_REF` environment variable, which must be a colon separated list of refs or globs.

  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`) and this variable is set to `true`, Git automatically copies your notes from the original to the rewritten commit. Defaults to `true`, but see "notes.rewriteRef" below.

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`, 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. The ref 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.

  This setting can be overridden with the `GIT_NOTES_REWRITE_REF` environment variable, which must be a colon separated list of refs or globs.

pack.window
: The size of the window used by 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 git-pack-objects(1) when no maximum depth is given on the command line. Defaults to 50.

pack.windowMemory
: The maximum size of memory that is consumed by each thread in 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 git-repack(1).

pack.deltaCacheSize
: The maximum memory in bytes used for caching deltas in 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 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.

pack.threads
: Specifies the number of threads to spawn when searching for best delta matches. This requires that 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 CPU's

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 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" and "rsync") 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 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 git-repack(1). 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.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, and that JGit's bitmap implementation does not understand it, causing it to complain if Git and JGit are used on the same repository. Defaults to false.

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 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.

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).

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 preserve, also pass `--preserve-merges` along to 'git rebase'
so that locally committed merge commits will not be flattened
by running 'git pull'.
```

  NOTE: this is a possibly dangerous operation; do **not** use it unless you understand the implications (see 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.default
: Defines the action `git push` should take if no refspec is explicitly given. 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 explicitly 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).

- `simple` - in centralized workflow, work like `upstream` with an added safety to refuse to push if the upstream branch's name is different from the local one.

  When pushing to a remote that is different from the remote you normally pull from, work as `current`. This is the safest option and is suited for beginners.

  This mode has become the default in Git 2.0.

- `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*.

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 *--autosquash* option by default.

rebase.autoStash
: When set to true, automatically create a temporary stash 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. Defaults to false.

receive.advertiseAtomic
: By default, git-receive-pack will advertise the atomic push capability to its clients. If you don't want to this capability to be advertised, set this variable to false.

receive.autogc
: By default, git-receive-pack will run "git-gc --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 verifies it by using a "nonce" protected by HMAC using this string as a secret key.

receive.certNonceSlop
: When a `git push --signed` sent 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. It will abort in the case of a malformed object or a broken link. The result of an abort are only dangling objects. Defaults to false. If not set, the value of `transfer.fsckObjects` is used instead.

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.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 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
: String(s) `receive-pack` uses to decide which refs to omit from its initial advertisement. Use more than one definitions to specify multiple prefix strings. A ref that are under the hierarchies listed on the value of this variable is excluded, and is hidden when responding to `git push`, and an attempt to update or delete a hidden ref by `git push` is rejected.

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 git-fetch(1) or git-push(1).

remote.<name>.pushurl
: The push URL of a remote repository. See 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>.fetch
: The default set of "refspec" for git-fetch(1). See git-fetch(1).

remote.<name>.push
: The default set of "refspec" for git-push(1). See 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 git-fetch(1) or the `update` subcommand of git-remote(1).

remote.<name>.skipFetchAll
: If true, this remote will be skipped by default when updating using git-fetch(1) or the `update` subcommand of git-remote(1).

remote.<name>.receivepack
: The default program to execute on the remote side when pushing. See option --receive-pack of git-push(1).

remote.<name>.uploadpack
: The default program to execute on the remote side when fetching. See option --upload-pack of 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 git-fetch(1) can override this setting. See options --tags and --no-tags of 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.

remotes.<group>
: The list of remotes which are fetched by "git remote update <group>". See git-remote(1).

repack.useDeltaBaseOffset

By default, 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 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.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. Defaults to false.

rerere.autoUpdate
: When set to true, `git-rerere` updates the index with the resulting contents after it cleanly resolves conflicts using previously recorded resolution. 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, 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.

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 git-send-email(1) for description. Note that this setting is not subject to the *identity* mechanism.

sendemail.smtpssl (deprecated)
: Deprecated alias for *sendemail.smtpEncryption = ssl*.

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 the this identity is selected, through command-line or *sendemail.identity*.

sendemail.aliasesFile

sendemail.aliasFileType

sendemail.annotate

sendemail.bcc

sendemail.cc

sendemail.ccCmd

sendemail.chainReplyTo

sendemail.confirm

sendemail.envelopeSender

sendemail.from

sendemail.multiEdit

sendemail.signedoffbycc

sendemail.smtpPass

sendemail.suppresscc

sendemail.suppressFrom

sendemail.to

sendemail.smtpDomain

sendemail.smtpServer

sendemail.smtpServerPort

sendemail.smtpServerOption

sendemail.smtpUser

sendemail.thread

sendemail.transferEncoding

sendemail.validate

sendemail.xmailer
: See git-send-email(1) for description.

sendemail.signedoffcc (deprecated)

Deprecated alias for *sendemail.signedoffbycc*.

showbranch.default
: The default set of branches for git-show-branch(1). See git-show-branch(1).

status.relativePaths
: By default, 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 git-status(1). The option --no-short takes precedence over this variable.

status.branch
: Set to true to enable --branch by default in git-status(1). The option --no-branch takes precedence over this variable.

status.displayCommentPrefix
: If set to true, 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 git-status(1) in Git 1.8.4 and previous. Defaults to false.

status.showUntrackedFiles
: By default, git-status(1) and 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 displays 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*. This variable can be overridden with the -u|--untracked-files option of git-status(1) and 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 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>.path

submodule.<name>.url
: The path within this project and URL for a submodule. These variables are initially populated by *git submodule init*. See git-submodule(1) and gitmodules(5) for details.

submodule.<name>.update
: The default update procedure for a submodule. This variable is populated by `git submodule init` from the gitmodules(5) file. See description of *update* command in 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 git-submodule(1) and 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 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 submodules 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.

tag.sort
: This variable controls the sort ordering of tags when displayed by git-tag(1). Without the "--sort=<value>" option provided, the value of this variable will be used as the default.

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 git-archive(1).

transfer.fsckObjects
: When `fetch.fsckObjects` or `receive.fsckObjects` are not set, the value of this variable is used instead. Defaults to false.

transfer.hideRefs
: This variable can be used to set both `receive.hideRefs` and `uploadpack.hideRefs` at the same time to the same values. See entries for these other variables.

transfer.unpackLimit
: When `fetch.unpackLimit` or `receive.unpackLimit` are not set, the value of this variable is used instead. The default value is 100.

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 git-upload-archive(1) for more details. Defaults to `false`.

uploadpack.hideRefs
: String(s) `upload-pack` uses to decide which refs to omit from its initial advertisement. Use more than one definitions to specify multiple prefix strings. A ref that are under the hierarchies listed on the value of this variable is excluded, and is hidden from `git ls-remote`, `git fetch`, etc. 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`.

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.

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.

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.email
: Your email address to be recorded in any newly created commits. Can be overridden by the *GIT_AUTHOR_EMAIL*, *GIT_COMMITTER_EMAIL*, and *EMAIL* environment variables. See git-commit-tree(1).

user.name
: Your full name to be recorded in any newly created commits. Can be overridden by the *GIT_AUTHOR_NAME* and *GIT_COMMITTER_NAME* environment variables. See git-commit-tree(1).

user.signingKey
: If git-tag(1) or 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.

versionsort.prereleaseSuffix
: When version sort is used in git-tag(1), prerelease tags (e.g. "1.0-rc1") may appear after the main release "1.0". By specifying the suffix "-rc" in this variable, "1.0-rc1" will appear before "1.0".

  This variable can be specified multiple times, once per suffix. The order of suffixes in the config file determines the sorting order (e.g. if "-pre" appears before "-rc" in the config file then 1.0-preXX is sorted before 1.0-rcXX). 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 git-instaweb(1) and git-help(1) may use it.

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-count-objects(1) Manual Page

NAME

git-count-objects - Count unpacked number of objects and their disk consumption

SYNOPSIS

> *git count-objects* [-v] [-H | --human-readable]

DESCRIPTION

This 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

Report in more detail:

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 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)

-H

--human-readable

Print sizes in human readable format

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-credential(1) Manual Page

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 the Git credential API 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 credential 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 contain the ones 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/password information (e.g. host, protocol, path), or to the actual credential data to be obtained (login/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. In both 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.

`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, from the user, or from a previously run helper).

`password`
: The credential's password, if we are asking it to be stored.

`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 any components which are missing from the URL (e.g., there is no username in the example above) will be set to empty; if you want to provide a URL and override some attributes, provide the URL attribute first, followed by any overrides.

Last updated 2014-11-27 19:56:10 CET

git-credential-cache(1) Manual Page

NAME

git-credential-cache - Helper to temporarily store passwords in memory

SYNOPSIS

```
git config credential.helper 'cache [options]'
```

DESCRIPTION

This command caches credentials in memory for use by future Git programs. The stored credentials never touch the disk, and are forgotten after a configurable timeout. 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 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 `~/.git-credential-cache/socket`. If your home directory is on a network-mounted filesystem, you may need to change this to a local filesystem.

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 drops the cache time to 5 minutes):

```
$ git config credential.helper 'cache --timeout=300'
```

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-credential-cache—daemon(1) Manual Page

NAME

git-credential-cache--daemon - Temporarily store user credentials in memory

SYNOPSIS

> git credential-cache—daemon [--debug] <socket>

DESCRIPTION

Note You probably don't want to invoke this command yourself; it is started automatically when you use git-credential-cache(1).

This command listens on the Unix domain socket specified by `<socket>` 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 git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-credential-store(1) Manual Page

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 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 gitcredentials(7) or `EXAMPLES` below.

OPTIONS

--file=<path>
: Use `<path>` to store credentials. The file will have its filesystem permissions set to prevent other users on the system from reading it, but will not be encrypted or otherwise protected. Defaults to `~/.git-credentials`.

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
```

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 gitcredentials(7) for more information.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-cvsexportcommit(1) Manual Page

NAME

git-cvsexportcommit - Export a single commit to a CVS checkout

SYNOPSIS

> *git cvsexportcommit* [-h] [-u] [-v] [-c] [-P] [-p] [-a] [-d cvsroot]
> [-w cvsworkdir] [-W] [-f] [-m msgprefix] [PARENTCOMMIT] COMMITID

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 *cvsexportcommit.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 git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-cvsimport(1) Manual Page

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 <commitlimit>]
[-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 or parsecvs.

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 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 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_EMAIL 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), `http://subversion.apache.org/`

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-cvsserver(1) Manual Page

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> ...]

OPTIONS

All these options obviously only make sense if enforced by the server side. They have been implemented to resemble the 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>
: You can specify a list of allowed directories. If no directories are given, all are allowed. This is an additional restriction, gitcvs access still needs to be enabled by the `gitcvs.enabled` config option unless *--export-all* was given, too.

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.

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 pserve 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:$1Oyx5r9mdGZ2
   myuser:$1$BA)@$vbnMJMDym7tA32AamXrm./
   ```

   You can use the *htpasswd* facility that comes with Apache to make these files, but Apache's MD5 crypt method differs from the one used by most C library's crypt() function, so don't use the -m option.

   Alternatively you can produce the password with perl's crypt() operator:

   ```
   perl -e 'my ($user, $pass) = @ARGV; printf "%s:%s\n", $user, crypt($user, $pass)' $USER password
   ```

   Then provide your password via the pserver method, for example:

   ```
   cvs -d:pserver:someuser:somepassword <at> 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 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_EMAIL, 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 can not 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:

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.

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 alpha-numeric 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 takes the place of the argument to --base-path.

GIT_CVSSERVER_ROOT specifies a single-directory whitelist. The repository must still be configured to allow access through git-cvsserver, as described above.

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.

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 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 git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-daemon(1) Manual Page

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>]]]
             [<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, you can further restrict the offers to a whitelist comprising of those.

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 whitelist is specified.

--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 whitelist.

--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. 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
: Log to syslog instead of stderr. Note that this option does not imply --verbose, thus by default only error conditions will be logged.

--user-path

--user-path=<path>
: Allow ~user notation to be used in requests. When specified with no parameter, requests 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>
: A directory to add to the whitelist of allowed directories. Unless --strict-paths is specified this will also include subdirectories of each named directory.

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 under the whitelisted set of directories, /pub/foo and /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 git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-describe(1) Manual Page

NAME

git-describe - Show the most recent tag that is reachable from a commit

SYNOPSIS

> *git describe* [--all] [--tags] [--contains] [--abbrev=<n>] <commit-ish>…
> *git describe* [--all] [--tags] [--contains] [--abbrev=<n>] --dirty[=<mark>]

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.

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 git-tag(1).

OPTIONS

<commit-ish>…
: Commit-ish object names to describe.

--dirty[=<mark>]
: Describe the working tree. It means describe HEAD and appends <mark> (`-dirty` by default) if the working tree is dirty.

--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 7 hexadecimal digits as 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. This can be used to avoid leaking private tags from the repository.

--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" + 7-char abbreviation for the tip commit of parent (which was `2414721b194453f058079d897d13c4e377f92dc6`). 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.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-diff(1) Manual Page

NAME

git-diff - Show changes between commits, commit and working tree, etc

SYNOPSIS

> *git diff* [options] [<commit>] [--] [<path>...]
> *git diff* [options] --cached [<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 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 git-add(1).

git diff --no-index [--options] [--] [<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.

git diff [--options] --cached [<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.

git diff [--options] <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.

git diff [--options] <commit> <commit> [--] [<path>...]
: This is to view the changes between two arbitrary <commit>.

git diff [--options] <commit>..<commit> [--] [<path>...]
: This is synonymous to the previous form. 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 if you are doing something exotic, it should be noted that all of the <commit> in the above description, except in the last two forms that use ".." notations, can be any <tree>.

For a more complete list of ways to spell <commit>, see "SPECIFYING REVISIONS" section in 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 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 section on generating patches). This is the default.

-s

--no-patch
: Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>
: Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw
: Generate the raw format.

--patch-with-raw
: Synonym for `-p --raw`.

--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.

--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 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. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (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>`.

--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.

--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 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`.

--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, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only
: Show only names of changed files.

--name-status
: Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]
: Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule(1) `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--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`.

--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.

The regex can also be set via a diff driver or configuration option, see gitattributes(1) or 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.

--check
: Warn if changes introduce whitespace errors. What are considered whitespace 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. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--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 only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. 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 lack 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 require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--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.

-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.

-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 !regexec(regexp, two->ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

  While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

  See the *pickaxe* entry in gitdiffcore(7) for more information.

--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>
: Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config(1)). To cancel `diff.orderFile`, use `-O/dev/null`.

-R
: Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]
: 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.

-a

--text
: Treat all files as text.

--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.

--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.

-W

--function-context
: Show whole surrounding functions of changes.

--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 gitattributes(5), you need to use this option with 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 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 git-diff(1) and git-log(1), but not for 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 git-config(1) or 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.

For more detailed explanation on these common options, see also gitdiffcore(7).

<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 creation, unmerged or "look at work tree".
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
- 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.

<sha1> is shown as all 0's if a file is new on the filesystem and it is out of sync with the index.

Example:

```
:100644 100644 5be4a4...... 000000...... M file.c
```

When `-z` option is not used, TAB, LF, and backslash characters in pathnames are represented as `\t`, `\n`, and `\\`, respectively.

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. single path, only for "dst"

Example:

```
::100644 100644 100644 fabadb8... cc95eb0... 4866510... MM	describe.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with 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 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 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 SHA-1 checksum 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. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.

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
   ```

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 git-diff(1) or git-show(1). Note also that you can give the `-m' option to any of these commands to force generation of diffs with individual parents of a merge.

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 with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--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 contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

   Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.

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 for apply. 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            <1>
$ git diff --cached   <2>
$ git diff HEAD       <3>
```

1. Changes in the working tree not yet staged for the next commit.
2. Changes between the index and your last commit; what you would be committing if you run "git commit" without "-a" option.
3. Changes in the working tree since your last commit; what you would be committing if you run "git commit -a"

Comparing with arbitrary commits

```
$ git diff test            <1>
$ git diff HEAD -- ./test  <2>
$ git diff HEAD^ HEAD      <3>
```

1. Instead of using the tip of the current branch, compare with the tip of "test" branch.
2. 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".
3. Compare the version before the last commit and the last commit.

Comparing branches

```
$ git diff topic master    <1>
$ git diff topic..master   <2>
$ git diff topic...master  <3>
```

1. Changes between the tips of the topic and the master branches.
2. Same as above.
3. 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            <1>
$ git diff --name-status                <2>
$ git diff arch/i386 include/asm-i386   <3>
```

1. Show only modification, rename, and copy, but not addition or deletion.
2. Show only names and the nature of change, but not actual diff output.
3. Limit diff output to named subtrees.

Munging the diff output

```
$ git diff --find-copies-harder -B -C  <1>
$ git diff -R                          <2>
```

1. Spend extra cycles to find renames, copies and complete rewrites (very expensive).
2. Output diff in reverse.

SEE ALSO

diff(1), git-difftool(1), git-log(1), gitdiffcore(7), git-format-patch(1), git-apply(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-diff-files(1) Manual Page

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 section on generating patches).

-s

--no-patch

Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw

Generate the raw format. This is the default.

--patch-with-raw

Synonym for `-p --raw`.

--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.

--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 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. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (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>`.

--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.

--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 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`.

--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, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only
: Show only names of changed files.

--name-status
: Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]
: Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule(1) `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--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`.

--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.

The regex can also be set via a diff driver or configuration option, see gitattributes(1) or 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.

--check
: Warn if changes introduce whitespace errors. What are considered whitespace 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. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--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 only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. 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 lack 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 require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the

specified number.

--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.

-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.

-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 !regexec(regexp, two->ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

  While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

  See the *pickaxe* entry in gitdiffcore(7) for more information.

--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>
: Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config(1)). To cancel `diff.orderFile`, use `-O/dev/null`.

-R
: Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]
: 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.

-a

--text
: Treat all files as text.

--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.

--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.

-W

--function-context
: Show whole surrounding functions of changes.

--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 gitattributes(5), you need to use this option with 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 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 git-diff(1) and git-log(1), but not for 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 git-config(1) or 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.

For more detailed explanation on these common options, see also 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 on 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 creation, unmerged or "look at work tree".
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
- 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.

<sha1> is shown as all 0's if a file is new on the filesystem and it is out of sync with the index.

Example:

```
:100644 100644 5be4a4...... 000000...... M file.c
```

When `-z` option is not used, TAB, LF, and backslash characters in pathnames are represented as `\t`, `\n`, and `\\`, respectively.

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. single path, only for "dst"

Example:

```
::100644 100644 100644 fabadb8... cc95eb0... 4866510... MM      describe.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with 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 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 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 SHA-1 checksum 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. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.

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
   ```

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 git-diff(1) or git-show(1). Note also that you can give the `-m' option to any of these commands to force generation of diffs with individual parents of a merge.

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 with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--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 contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.
3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

   Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.
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 for apply. 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 git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-diff-index(1) Manual Page

NAME

git-diff-index - Compare a tree to the working tree or index

SYNOPSIS

> *git diff-index* [-m] [--cached] [<common diff options>] <tree-ish> [<path>...]

DESCRIPTION

Compares 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, compares only paths matching those patterns. Otherwise all tracked files are compared.

OPTIONS

-p

-u

--patch
: Generate patch (see section on generating patches).

-s

--no-patch
: Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>
: Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw
: Generate the raw format. This is the default.

--patch-with-raw
: Synonym for `-p --raw`.

--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.

--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 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. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (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>`.

--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.

--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 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`.

--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, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only
: Show only names of changed files.

--name-status
: Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]
: Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule(1) `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--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`.

--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.

  The regex can also be set via a diff driver or configuration option, see gitattributes(1) or 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.

--check
: Warn if changes introduce whitespace errors. What are considered whitespace 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. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--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 only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. 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 lack 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 require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--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.

-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.

-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 !regexec(regexp, two->ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

See the *pickaxe* entry in gitdiffcore(7) for more information.

--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>
: Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config(1)). To cancel `diff.orderFile`, use `-O/dev/null`.

-R
: Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]
: 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.

-a

--text
: Treat all files as text.

--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.

--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.

-W

--function-context
: Show whole surrounding functions of changes.

--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 gitattributes(5), you need to use this option with 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 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 git-diff(1) and git-log(1), but not for 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 git-config(1) or 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.

For more detailed explanation on these common options, see also gitdiffcore(7).

<tree-ish>
: The id of a tree object to diff against.

--cached
: do not consider the on-disk file at all

-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 creation, unmerged or "look at work tree".
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
- 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.

<sha1> is shown as all 0's if a file is new on the filesystem and it is out of sync with the index.

Example:

```
:100644 100644 5be4a4...... 000000...... M file.c
```

When `-z` option is not used, TAB, LF, and backslash characters in pathnames are represented as `\t`, `\n`, and `\\`, respectively.

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. single path, only for "dst"

Example:

```
::100644 100644 100644 fabadb8... cc95eb0... 4866510... MM      describe.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with 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 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 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 SHA-1 checksum 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. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.

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
   ```

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 git-diff(1) or git-show(1). Note also that you can give the `-m' option to any of these commands to force generation of diffs with individual parents of a merge.

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 with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--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 contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.
3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

   Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.
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 for apply. 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 blob    4161aecc6700a2eb579e842af0b7f22b98443f74        commit.c
+100644 blob    4161aecc6700a2eb579e842af0b7f22b98443f74        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 100664 7476bb... 000000...      kernel/sched.c
```

i.e., it shows that the tree has changed, and that `kernel/sched.c` has 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 git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-difftool(1) Manual Page

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 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.

-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 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

--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`.

--[no-]trust-exit-code
: *git-difftool* invokes a diff tool individually on each file. 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 git-diff(1) for the full list of supported options.

CONFIG VARIABLES

git difftool falls back to *git mergetool* config variables when the difftool equivalents have not been defined.

diff.tool
: The default diff tool to use.

diff.guitool
: The default diff tool to use when `--gui` is specified.

difftool.<tool>.path
: Override the path for the given tool. This is useful in case your tool is not in the PATH.

difftool.<tool>.cmd
: Specify the command to invoke the specified diff tool.

  See the `--tool=<tool>` option above for more details.

difftool.prompt
: Prompt before each invocation of the diff tool.

difftool.trustExitCode
: Exit difftool if the invoked diff tool returns a non-zero exit status.

  See the `--trust-exit-code` option above for more details.

SEE ALSO

git-diff(1)
: Show changes between commits, commit and working tree, etc

git-mergetool(1)
: Run merge conflict resolution tools to resolve merge conflicts

git-config(1)
: Get and set repository or global options

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-diff-tree(1) Manual Page

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] [--root] [<common diff options>]
>   <tree-ish> [<tree-ish>] [<path>...]

DESCRIPTION

Compares the content and mode of the 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 section on generating patches).

-s

--no-patch
: Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>
: Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw
: Generate the raw format. This is the default.

--patch-with-raw
: Synonym for `-p --raw`.

--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.

--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 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. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (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>`.

--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.

--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 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`.

--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, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only
: Show only names of changed files.

--name-status
: Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]
: Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule(1) `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--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`.

--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.

The regex can also be set via a diff driver or configuration option, see gitattributes(1) or 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.

--check
: Warn if changes introduce whitespace errors. What are considered whitespace 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. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--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 only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. 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 lack 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 require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--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.

-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.

-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 !regexec(regexp, two->ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

  While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

  See the *pickaxe* entry in gitdiffcore(7) for more information.

--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>
: Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config(1)). To cancel `diff.orderFile`, use `-O/dev/null`.

-R
: Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]
: 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.

-a

--text
: Treat all files as text.

--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.

--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.

-W

--function-context
: Show whole surrounding functions of changes.

--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 gitattributes(5), you need to use this option with 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 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 git-diff(1) and git-log(1), but not for 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 git-config(1) or 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.

For more detailed explanation on these common options, see also 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 these prefix strings. i.e., file matches `/^<pattern1>|<pattern2>|.../` Note that this parameter does not provide any wildcard or regexp features.

-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.

--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 *-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*, *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 git-config(1)).

--abbrev-commit
: Instead of showing the full 40-byte hexadecimal commit object name, show only a partial prefix. Non default number of digits can be specified with "--abbrev=<n>" (which also modifies diff output, if it is displayed).

  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` and those options which imply it such as "--oneline". It also overrides the *log.abbrevCommit* variable.

--oneline
: This is a shorthand for "--pretty=oneline --abbrev-commit" used together.

--encoding=<encoding>
: The commit objects record the 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.

--notes[=<ref>]
: Show the notes (see 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 git-config(1) for more details.

  With an optional *<ref>* argument, show this notes ref instead of the default notes ref(s). The ref is taken to be in `refs/notes/` if it is not qualified.

  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[=<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 the 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 is not shown, just like in any other "empty diff" case.

--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 sha1s 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 git-config(1)). Here are the details of the built-in formats:

- *oneline*

```
<sha1> <title line>
```

This is designed to be as compact as possible.

- *short*

  ```
  commit <sha1>
  Author: <author>

  <title line>
  ```

- *medium*

  ```
  commit <sha1>
  Author: <author>
  Date:   <author date>

  <title line>

  <full commit message>
  ```

- *full*

  ```
  commit <sha1>
  Author: <author>
  Commit: <committer>

  <title line>

  <full commit message>
  ```

- *fuller*

  ```
  commit <sha1>
  Author:     <author>
  AuthorDate: <author date>
  Commit:     <committer>
  CommitDate: <committer date>

  <title line>

  <full commit message>
  ```

- *email*

  ```
  From <sha1> <date>
  From: <author>
  Date: <author date>
  Subject: [PATCH] <title line>

  <full commit message>
  ```

- *raw*

  The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the SHA-1s 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.

- *format:<string>*

  The *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:

  - *%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 git-shortlog(1) or git-blame(1))
- *%ae*: author email
- *%aE*: author email (respecting .mailmap, see git-shortlog(1) or 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
- *%cn*: committer name
- *%cN*: committer name (respecting .mailmap, see git-shortlog(1) or git-blame(1))
- *%ce*: committer email
- *%cE*: committer email (respecting .mailmap, see git-shortlog(1) or 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
- *%d*: ref names, like the --decorate option of git-log(1)
- *%D*: ref names without the " (", ")" wrapping.
- *%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 signature, "B" for a Bad signature, "U" for a good, untrusted signature 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
- *%gD*: reflog selector, e.g., `refs/stash@{1}`
- *%gd*: shortened reflog selector, e.g., `stash@{1}`
- *%gn*: reflog identity name
- *%gN*: reflog identity name (respecting .mailmap, see git-shortlog(1) or git-blame(1))
- *%ge*: reflog identity email
- *%gE*: reflog identity email (respecting .mailmap, see git-shortlog(1) or git-blame(1))
- *%gs*: reflog subject
- *%Cred*: switch color to red
- *%Cgreen*: switch color to green
- *%Cblue*: switch color to blue
- *%Creset*: reset color
- *%C(…)*: color specification, as described in color.branch.* config option; adding `auto,` at the beginning will emit color only when colors are 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). `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
- *%n*: newline
- *%%*: a raw *%*
- *%x00*: print a byte from a hex code
- *%w([<w>[,<i1>[,<i2>]]])*: switch line wrapping, like the -w option of git-shortlog(1).

- *%<(<N>[,trunc|ltrunc|mtrunc])*: make the next placeholder take at least N columns, padding spaces on the right if necessary. Optionally truncate at the beginning (ltrunc), the middle (mtrunc) or the end (trunc) if the output is longer than N columns. Note that truncating only works correctly with N >= 2.
- *%<|(<N>)*: make the next placeholder take at least until Nth columns, padding spaces on the right if necessary
- *%>(<N>)*, *%>|(<N>)*: similar to *%<(<N>)*, *%<|(<N>)* respectively, but padding spaces on the left
- *%>>(<N>)*, *%>>|(<N>)*: similar to *%>(<N>)*, *%>|(<N>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces
- *%><(<N>)*, *%><|(<N>)*: similar to *% <(<N>)*, *%<|(<N>)* respectively, but padding both sides (i.e. the text is centered)

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.

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, line-feeds that immediately precede 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
```

Limiting Output

If you're only interested in differences in a subset of files, for example some architecture-specific files, you might do:

```
git diff-tree -r <tree-ish> <tree-ish> arch/ia64 include/asm-ia64
```

and it will only show you what changed in those two directories.

Or if you are searching for what changed in just `kernel/sched.c`, just do

```
git diff-tree -r <tree-ish> <tree-ish> kernel/sched.c
```

and it will ignore all differences to other files.

The pattern is always the prefix, and is matched exactly. There are no wildcards. Even stricter, it has to match a complete path component. I.e. "foo" does not pick up `foobar.h`. "foo" does match `foo/bar.h` so it can be used to name subdirectories.

An example of normal usage is:

```
torvalds@ppc970:~/git> git diff-tree --abbrev 5319e4
:100664 100664 ac348b... a01513...    git-fsck-objects.c
```

which tells you that the last commit changed just one file (it's from this one:

```
commit 3c6f7ca19ad4043e9e72fa94106f352897e651a8
tree 5319e4d609cdd282069cc4dce33c1db559539b03
parent b4e628ea30d5ab3606119d2ea5caeab141d38df7
author Linus Torvalds <torvalds@ppc970.osdl.org> Sat Apr 9 12:02:30 2005
committer Linus Torvalds <torvalds@ppc970.osdl.org> Sat Apr 9 12:02:30 2005

Make "git-fsck-objects" print out all the root commits it finds.

Once I do the reference tracking, I'll also make it print out all the
HEAD commits it finds, which is even more interesting.
```

in case you care).

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 creation, unmerged or "look at work tree".
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

- 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.

<sha1> is shown as all 0's if a file is new on the filesystem and it is out of sync with the index.

Example:

```
:100644 100644 5be4a4...... 000000...... M file.c
```

When `-z` option is not used, TAB, LF, and backslash characters in pathnames are represented as `\t`, `\n`, and `\\`, respectively.

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. single path, only for "dst"

Example:

```
::100644 100644 100644 fabadb8... cc95eb0... 4866510... MM      describe.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with 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 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 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 SHA-1 checksum 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. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.
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
```

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 git-diff(1) or git-show(1). Note also that you can give the `-m' option to any of these commands to force generation of diffs with individual parents of a merge.

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 with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--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 contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.
3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

   Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.
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 for apply. 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 git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-fast-export(1) Manual Page

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 git-bundle(1)), or as a kind of an interactive *git filter-branch*.

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 git-rev-list(1))

-M

-C
: Perform move and/or copy detection, as described in the 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.

Any commits 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.

--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.

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.

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

git-fast-import(1)

GIT

Part of the git(1) suite

Last updated 2014-12-13 19:39:10 CET

git-fast-import(1) Manual Page

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 all non-fatal output, making fast-import silent when it is successful. This option disables the output shown by --stats.

--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.

Options for Frontends

--cat-blob-fd=<fd>
: Write responses to `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.

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.

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 10.

--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.

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.

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.

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.

`rfc2822`

This is the standard email 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`.

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.

`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.

`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.

commit

Create or update a branch with a new commit, recording one logical change to the project.

```
        'commit' SP <ref> LF
        mark?
        ('author' (SP <name>)? SP LT <email> GT SP <when> LF)?
        'committer' (SP <name>)? SP LT <email> GT SP <when> LF
        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).

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`.

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.

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 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.

`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).

`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 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>` string must use UNIX-style directory separators (forward slash `/`), may contain any byte other than `LF`, and must not start with double quote (`"`).

A path can use C-style string quoting; this is accepted in all cases and mandatory if the filename starts with double quote or contains `LF`. In C-style quoting, the complete name should be surrounded with double quotes, and any `LF`, backslash, or double quote characters must be escaped by preceding them with a backslash (e.g., `"path/with\n, \\ and \" in it"`).

The value of `<path>` 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>`.

It is recommended that `<path>` always be encoded using UTF-8.

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>`.

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.

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`.

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.

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).

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.

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
        'from' SP <commit-ish> LF
        '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.

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.

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?
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.

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).

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).

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.

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 anywhere in the stream that comments are accepted. In particular, the `cat-blob` command can be used in the middle of a commit but not in the middle of a `data` command.

See "Responses To Commands" below for details about how to read this output safely.

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 be used anywhere in the stream that comments are accepted, including 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 *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.

`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.

cat-blob

ls
: Require that the backend support the *cat-blob* or *ls* command. 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.

`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

`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`, `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`, 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.

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.

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.

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.

Use Tag Fixup Branches

Some other SCM systems let the user create a tag from multiple files which are not from the same commit/changeset. 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.

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.

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.

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).

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.

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.

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.

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=.

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.

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.

SEE ALSO

git-fast-export(1)

GIT

Part of the git(1) suite

Last updated 2015-05-03 21:16:24 CEST

git-fetch(1) Manual Page

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 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 git-pull(1).

OPTIONS

--all
: Fetch all remotes.

-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.

--depth=<depth>
: Deepen or shorten the history of a *shallow* repository created by `git clone` with `--depth=<depth>` option (see git-clone(1)) to the specified number of commits from the tip of each remote branch history. Tags for the deepened commits are not fetched.

--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 accept such refs.

--dry-run
: Show what would be done, without making any changes.

-f

--force
: When *git fetch* is used with `<rbranch>:<lbranch>` refspec, it refuses to update the local branch `<lbranch>` unless the remote branch `<rbranch>` it fetches is a descendant of `<lbranch>`. 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.

-p

--prune
: After 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.

-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 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. 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 populated submodules should be fetched too. It can be used as a boolean option to completely disable recursion when set to *no* or to unconditionally recurse into all populated submodules when set to *yes*, which is the default when this option is used without any value. Use *on-demand* to only recurse into a populated submodule when the superproject retrieves a commit that updates the submodule's reference to a commit that isn't already in the local submodule clone.

--no-recurse-submodules
: Disable recursive fetching of submodules (this has the same effect as using the *--recurse-submodules=no* option).

--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 gitmodules(5) and 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.

<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 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 ref <src>, followed by a colon `:`, followed by the destination ref <dst>. The colon can be omitted when <dst> is empty.

  `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 empty string, the local ref that matches it is fast-forwarded using <src>. If the optional plus `+` is used, the local ref is updated even if it does not result in a fast-forward update.

  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 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.

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 and rsync can be used for fetching and pushing, but these are 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.xz[:port]/path/to/repo.git/
- git://host.xz[:port]/path/to/repo.git/
- http[s]://host.xz[:port]/path/to/repo.git/
- ftp[s]://host.xz[:port]/path/to/repo.git/
- rsync://host.xz/path/to/repo.git/

An alternative scp-like syntax may also be used with the ssh protocol:

- [user@]host.xz:path/to/repo.git/

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.xz[:port]/~[user]/path/to/repo.git/
- git://host.xz[:port]/~[user]/path/to/repo.git/
- [user@]host.xz:/~[user]/path/to/repo.git/

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 git-clone(1) for details.

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 gitremote-helpers(1) 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.

Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using git-remote(1), 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>`.

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 format
        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.

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.

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 branch.<name>.fetch option is used to specify a non-default refspec.
- Using refspecs explicitly:

  ```
  $ git fetch origin +pu:pu maint:tmp
  ```

  This updates (or creates, as necessary) branches `pu` and `tmp` in the local repository by fetching from the branches (respectively) `pu` and `maint` from the remote repository.

  The `pu` branch will be updated even if it is 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 git-log(1). The fetched objects will eventually be removed by git's built-in housekeeping (see git-gc(1)).

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 can not 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

git-pull(1)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-fetch-pack(1) Manual Page

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 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.

--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.

SEE ALSO

git-fetch(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-filter-branch(1) Manual Page

NAME

git-filter-branch - Rewrite branches

SYNOPSIS

```
git filter-branch [--env-filter <command>] [--tree-filter <command>]
        [--index-filter <command>] [--parent-filter <command>]
        [--msg-filter <command>] [--commit-filter <command>]
        [--tag-name-filter <command>] [--subdirectory-filter <directory>]
        [--prune-empty]
        [--original <namespace>] [-d <directory>] [-f | --force]
        [--] [<rev-list options>...]
```

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 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.

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 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

--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 git-commit-tree(1) for details). Do not forget to re-export the variables.

--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 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 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.

--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 [Remap_to_ancestor].

--prune-empty
: Some kind of filters will generate empty commits, that left the tree untouched. This switch allow git-filter-branch to ignore such commits. Though, this switch only applies for commits that have one and only one parent, it will hence keep merges points. Also, this option is not compatible with the use of *--commit-filter*. Though you just need to use the function *git_commit_non_empty_tree "$@"* instead of the `git commit-tree "$@"` idiom in your commit filter to make that happen.

--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.

<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 [Remap_to_ancestor].

Remap to ancestor

By using 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.

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:

```
echo "$commit-id $graft-id" >> .git/info/grafts
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 "$GIT_AUTHOR_EMAIL" = "root@localhost"
        then
                GIT_AUTHOR_EMAIL=john@example.com
                export GIT_AUTHOR_EMAIL
        fi
        if test "$GIT_COMMITTER_EMAIL" = "root@localhost"
        then
```

```
                GIT_COMMITTER_EMAIL=john@example.com
                export GIT_COMMITTER_EMAIL
        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 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).

Notes

git-filter-branch allows you to make complex shell-scripted rewrites of your Git history, but you probably don't need this flexibility if you're simply *removing unwanted data* like large files or passwords. For those operations you may want to consider The BFG Repo-Cleaner, a JVM-based alternative to git-filter-branch, typically at least 10-50x faster for those use-cases, and with quite different characteristics:

- Any particular version of a file is cleaned exactly *once*. The BFG, unlike git-filter-branch, does not give you the opportunity to handle a file differently based on where or when it was committed within your history. This constraint gives the core performance benefit of The BFG, and is well-suited to the task of cleansing bad data - you don't care *where* the bad data is, you just want it *gone*.
- By default The BFG takes full advantage of multi-core machines, cleansing commit file-trees in parallel. git-filter-branch cleans commits sequentially (i.e. in a single-threaded manner), though it *is* possible to write filters that include their own parallelism, in the scripts executed against each commit.

- The command options are much more restrictive than git-filter branch, and dedicated just to the tasks of removing unwanted data- e.g: `--strip-blobs-bigger-than 1M`.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-fmt-merge-msg(1) Manual Page

NAME

git-fmt-merge-msg - Produce a merge commit message

SYNOPSIS

> *git fmt-merge-msg* [-m <message>] [--log[=<n>] | --no-log] <$GIT_DIR/FETCH_HEAD
> *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`.

-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.summary

Synonym to `merge.log`; this is deprecated and will be removed in the future.

SEE ALSO

git-merge(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:55:05 CET

git-for-each-ref(1) Manual Page

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>] [<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

<count>

By default the command shows all refs that match `<pattern>`. This option makes it stop after showing that many refs.

<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>

A string that interpolates `%(fieldname)` from the object pointed at by a ref being shown. If `fieldname` is prefixed with an asterisk (`*`) and the ref points at a tag object, the value for the field in the object tag refers is used. When unspecified, defaults to `%(objectname) SPC %(objecttype) TAB %(refname)`. 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).

<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.

--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.

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.

objecttype
: The type of the object (`blob`, `tree`, `commit`, `tag`).

objectsize
: The size of the object (the same as *git cat-file -s* reports).

objectname
: The object name (aka SHA-1). For a non-ambiguous abbreviation of the object name append `:short`.

upstream
: The name of a local ref which can be considered “upstream” from the displayed ref. Respects `:short` 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). Has no effect if the ref does not have tracking information associated with it.

HEAD
: * if HEAD matches current ref (the checked out branch), ' ' otherwise.

color
: Change output color. Followed by `:<colorname>`, where names are described in `color.branch.*`.

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 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.

The complete message in a commit and tag object is `contents`. Its first line is `contents:subject`, where subject is the concatenation of all lines of the commit message up to the first blank line. The next line is *contents:body*, where body is all of the lines after the first blank line. Finally, the optional GPG signature is `contents:signature`.

For sorting purposes, fields with numeric values sort in numeric order (`objectsize`, `authordate`, `committerdate`, `taggerdate`). All other fields are used to sort in their byte-value order.

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 one of `:default`, `:relative`, `:short`, `:local`, `:iso8601`, `:rfc2822` or `:raw` to the end of the fieldname; e.g. `%(taggerdate:relative)`.

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"
```

SEE ALSO

git-show-ref(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-format-patch(1) Manual Page

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]
                 [--subject-prefix=Subject-Prefix] [(--reroll-count|-v) <n>]
                 [--to=<email>] [--cc=<email>]
                 [--[no-]cover-letter] [--quiet] [--notes[=<ref>]]
                 [<common diff options>]
                 [ <since> | <revision range> ]
```

DESCRIPTION

Prepare each commit with its patch in one file per commit, formatted to resemble UNIX mailbox format. The output of this command is convenient for e-mail submission or for use with *git am*.

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 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.

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 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.

-s

--no-patch
: Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>
: Generate diffs with <n> lines of context instead of the usual three.

--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.

--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 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. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (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>`.

--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.

--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 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`.

--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.

--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 only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. 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 lack 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 require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

-O<orderfile>
: Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config(1)). To cancel `diff.orderFile`, use `-O/dev/null`.

-a

--text
: Treat all files as text.

--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.

--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.

-W

--function-context
: Show whole surrounding functions of changes.

--ext-diff
: Allow an external diff helper to be executed. If you set an external diff driver with gitattributes(5), you need to use this option with 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 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 git-diff(1) and git-log(1), but not for 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 git-config(1) or 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.

For more detailed explanation on these common options, see also 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 `Signed-off-by:` line to the commit message, using the committer identity of yourself.

--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. If `--thread` is specified without a style, it defaults to the style specified by *format.thread* if any, or else `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.

--subject-prefix=<Subject-Prefix>
: Instead of the standard *[PATCH]* prefix in the subject line, instead use *[<Subject-Prefix>]*. This allows for useful naming of a patch series, and can be combined with the `--numbered` option.

-v <n>

--reroll-count=<n>
: Mark the series as the <n>-th iteration of the topic. The output filenames have `v<n>` pretended 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.

--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`.

--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 shortlog and the overall diffstat. You can fill in a description in the file before sending it out.

--notes[=<ref>]
: Append the notes (see 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 git-notes(1) to use this workflow).

--[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.

--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.

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, 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
        coverletter = 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 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.

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 checkout test-apply
  $ git reset --hard
  $ 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 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.

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 git-send-email(1).

For hints on submission using the IMAP interface, see the EXAMPLE section of git-imap-send(1).

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.

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.

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.

Approach #3 (external editor)

The following Thunderbird extensions are needed: AboutConfig from http://aboutconfig.mozdev.org/ and External Editor from http://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.

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.

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
  ```

SEE ALSO

git-am(1), git-send-email(1)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-fsck(1) Manual Page

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] [<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 `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.

--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 triggers 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.

--[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.

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).

Extracted Diagnostics

expect dangling commits - potential heads - due to lack of head information
: You haven't specified any nodes as heads so it won't be possible to differentiate between un-parented commits and root nodes.

missing sha1 directory <*dir*>
: The directory holding the sha1 objects is missing.

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.

sha1 mismatch <object>
: The database has an object who's sha1 doesn't match the database value. This indicates a serious data integrity problem.

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 git(1) suite

Last updated 2014-11-27 19:55:05 CET

git-fsck-objects(1) Manual Page

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 git-fsck(1). Please refer to the documentation of that command.

GIT

Part of the git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-gc(1) Manual Page

NAME

git-gc - Cleanup unnecessary files and optimize the local repository

SYNOPSIS

> *git gc* [--aggressive] [--auto] [--quiet] [--prune=<date> | --no-prune] [--force]

DESCRIPTION

Runs a number of housekeeping tasks within the current repository, such as compressing file revisions (to reduce disk space and increase performance) and removing unreachable objects which may have been created from prior invocations of *git add*.

Users are encouraged to run this task on a regular basis within each repository to maintain good disk space utilization and good operating performance.

Some git commands may automatically run *git gc*; see the `--auto` flag below for details. If you know what you're doing and all you want is to disable this behavior permanently without further considerations, just do:

```
$ git config --global gc.auto 0
```

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 persistent, so this option only needs to be used occasionally; every few hundred changesets or so.

--auto
: With this option, *git gc* checks whether any housekeeping is required; if not, it exits without performing any work. Some git commands run `git gc --auto` after performing operations that could create many loose objects.

  Housekeeping is required if there are too many loose objects or too many packs in the repository. If the number of loose objects exceeds the value of the `gc.auto` configuration variable, then all loose objects are combined into a single pack using `git repack -d -l`. Setting the value of `gc.auto` to 0 disables automatic packing of loose objects.

  If the number of packs exceeds the value of `gc.autoPackLimit`, then existing packs (except those marked with a `.keep` file) are consolidated into a single pack by using the `-A` option of *git repack*. Setting `gc.autoPackLimit` to 0 disables automatic consolidation of packs.

--prune=<date>
: Prune loose objects older than date (default is 2 weeks ago, overridable by the config variable `gc.pruneExpire`). --prune=all prunes loose objects regardless of their age. --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.

Configuration

The optional configuration variable *gc.reflogExpire* can be set to indicate how long historical entries within each branch's reflog should remain available in this repository. The setting is expressed as a length of time, for example *90 days* or *3 months*. It defaults to *90 days*.

The optional configuration variable *gc.reflogExpireUnreachable* can be set to indicate how long historical reflog entries which are not part of the current branch should remain available in this repository. 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. This option defaults to *30 days*.

The above two configuration variables can be given to a pattern. For example, this sets non-default expiry values only to remote-tracking branches:

```
[gc "refs/remotes/*"]
        reflogExpire = never
        reflogExpireUnreachable = 3 days
```

The optional configuration variable *gc.rerereResolved* indicates how long records of conflicted merge you resolved earlier are kept. This defaults to 60 days.

The optional configuration variable *gc.rerereUnresolved* indicates how long records of conflicted merge you have not resolved are kept. This defaults to 15 days.

The optional configuration variable *gc.packRefs* determines if *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. This defaults to true.

The optional configuration variable *gc.aggressiveWindow* controls how much time is spent optimizing the delta compression of the objects in the repository when the --aggressive option is specified. The larger the value, the more time is spent optimizing the delta compression. See the documentation for the --window' option in git-repack(1) for more details. This defaults to 250.

Similarly, the optional configuration variable *gc.aggressiveDepth* controls --depth option in git-repack(1). This defaults to 250.

The optional configuration variable *gc.pruneExpire* controls how old the unreferenced loose objects have to be before they are pruned. The default is "2 weeks ago".

Notes

git gc tries very hard to be safe about the garbage it collects. 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, refs saved by *git filter-branch* in refs/original/, or reflogs (which may reference commits in branches that were later amended or rewound).

If you are expecting some objects to be collected and they aren't, check all of those locations and decide whether it makes sense in your case to remove those references.

HOOKS

The *git gc --auto* command will run the *pre-auto-gc* hook. See githooks(5) for more information.

SEE ALSO

git-prune(1) git-reflog(1) git-repack(1) git-rerere(1)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-get-tar-commit-id(1) Manual Page

NAME

git-get-tar-commit-id - Extract commit ID from an archive created using git-archive

SYNOPSIS

> *git get-tar-commit-id* < <tarfile>

DESCRIPTION

Acts as a filter, extracting the commit ID stored in archives created by *git archive*. It reads only the first 1024 bytes of input, thus its runtime is not influenced by the size of <tarfile> very much.

If no commit ID is found, *git get-tar-commit-id* quietly exists with a return code of 1. This can happen if <tarfile> 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 git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-grep(1) Manual Page

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]
           [-l | --files-with-matches] [-L | --files-without-match]
           [(-O | --open-files-in-pager) [<pager>]]
           [-z | --null]
           [-c | --count] [--all-match] [-q | --quiet]
           [--max-depth <depth>]
           [--color[=<when>] | --no-color]
           [--break] [--heading] [-p | --show-function]
           [-A <post-context>] [-B <pre-context>] [-C <context>]
           [-W | --function-context]
           [-f <file>] [-e] <pattern>
           [--and|--or|--not|(|)|-e <pattern>...]
           [ [--[no-]exclude-standard] [--cached | --no-index | --untracked] | <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.

CONFIGURATION

grep.lineNumber
: If set to true, enable *-n* 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 return to the default matching behavior.

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.fullName
: If set to true, enable *--full-name* option by default.

OPTIONS

--cached
: Instead of searching tracked files in the working tree, search blobs registered in the index file.

--no-index
: Search files in the current directory that is not managed by Git.

--untracked
: In addition to searching in the tracked files in the working tree, search also in untracked files.

--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`.

-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 negative value 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.

-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 regexp for patterns. Requires libpcre to be compiled in.

-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.

-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.

-z

--null
: Output \0 instead of the character that normally follows a file name.

-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 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.

-f <file>
: Read patterns from <file>, one per line.

-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.

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

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-gui(1) Manual Page

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 (or *HEAD* by default). 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 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

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.

A *git gui* development repository can be obtained from:

```
git clone git://repo.or.cz/git-gui.git
```

or

```
git clone http://repo.or.cz/r/git-gui.git
```

or browsed online at http://repo.or.cz/w/git-gui.git/.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-hash-object(1) Manual Page

NAME

git-hash-object - Compute object ID and optionally creates a blob from a file

SYNOPSIS

> *git hash-object* [-t <type>] [-w] [--path=<file>|--no-filters] [--stdin] [--] <file>...
> *git hash-object* [-t <type>] [-w] --stdin-paths [--no-filters] < <list-of-paths>

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. This is used by *git cvsimport* to update the index without modifying files in the work tree. When <type> is not specified, it defaults to "blob".

OPTIONS

-t <type>
: Specify the type (default: "blob").

-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 stdin instead of from the command-line.

--path
: Hash object as it were located at the given path. The location of file does not directly influence on the hash value, but path is used to determine what Git filters should be applied to the object before it can be placed to the object database, and, as 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.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-help(1) Manual Page

NAME

git-help - Display help information about Git

SYNOPSIS

> *git help* [-a|--all] [-g|--guide]
>   [-i|--info|-m|--man|-w|--web] [COMMAND|GUIDE]

DESCRIPTION

With no options and no COMMAND or GUIDE 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 *--guide* or *-g* is given, a list of the useful Git guides is also printed on the standard output.

If a command, or a guide, is given, a manual page for that command or guide is brought up. The *man* program is used by default for this purpose, but this can be overridden by other options or configuration variables.

Note that `git --help ...` is identical to `git help ...` because the former is internally converted into the latter.

To display the git(1) man page, use `git help git`.

This page can be displayed with *git help help* or `git help --help`

OPTIONS

-a

--all
: Prints all the available commands on the standard output. This option overrides any given command or guide name.

-g

--guides
: Prints a list of useful guides on the standard output. This option overrides any given command or guide name.

-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 none of these config variables is set, the *git web--browse* helper script (called by *git help*) will pick a suitable default. See git-web--browse(1) for more information about this.

CONFIGURATION VARIABLES

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*.

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 git-web--browse(1).

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.

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.

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.

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
```

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 git-config(1) for more information about this.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-http-backend(1) Manual Page

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 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 environmental 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 item 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 item 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/
```

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 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 environmental 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 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 git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-http-fetch(1) Manual Page

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] <commit> <url>

DESCRIPTION

Downloads a remote Git repository via HTTP.

NOTE: use of this command without -a is deprecated. The -a behaviour will become the default in a future release.

OPTIONS

commit-id
: Either the hash or the filename under [URL]/refs/ to pull.

-c
: Get the commit objects.

-t
: Get trees associated with the commit objects.

-a
: Get all the objects.

-v
: Report what is downloaded.

-w <filename>
: Writes the commit-id into the 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>]
```

--recover
: Verify that everything reachable from target is fetched. Used after an earlier fetch is interrupted.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-http-push(1) Manual Page

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 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 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 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 in order to avoid accidentally overwriting the remote ref and lose 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 git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-imap-send(1) Manual Page

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.

Variables

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 setup 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 a `imap://` prefix for non-secure connections and a `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 authenticate method for authentication with 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
```

EXAMPLE

To submit patches using GMail's IMAP interface, first, edit your ~/.gitconfig to specify your account settings:

```
[imap]
        folder = "[Gmail]/Drafts"
        host = imaps://imap.gmail.com
        user = user@gmail.com
        port = 993
        sslverify = false
```

You might need to instead use: folder = "[Google Mail]/Drafts" if you get an error that the "Folder doesn't exist".

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: http://kb.mozillazine.org/Plain_text_e-mail_-_Thunderbird#Completely_plain_email

SEE ALSO

git-format-patch(1), git-send-email(1), mbox(5)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-index-pack(1) Manual Page

NAME

git-index-pack - Build pack index file for an existing packed archive

SYNOPSIS

> *git index-pack* [-v] [-o <index-file>] <pack-file>
> *git index-pack* --stdin [--fix-thin] [--keep] [-v] [-o <index-file>]
>   [<pack-file>]

DESCRIPTION

Reads a packed archive (.pack) from the specified file, and builds a pack index file (.idx) for it. 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).

--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 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, but rather than 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
: Die, if the pack contains broken objects or links.

--check-self-contained-and-connected
: Die if the pack contains broken links. For internal use only.

--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.

Note

Once the index has been created, the list of object names is sorted and the SHA-1 hash of that list 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 git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-init(1) Manual Page

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>]
[--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 `HEAD` file that references the HEAD of the master branch is also created.

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.

--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 filesystem-agnostic Git symbolic link to the repository.

  If this is reinitialization, the repository will be moved to the specified path.

--shared[=(false|true|umask|group|all|world|everybody|0xxx)]
: 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 (or *false*)
  : Use permissions reported by umask(2). The default, when `--shared` is not specified.

  group (or *true*)
  : Make the repository group-writable, (and g+sx, since the git group may be not 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 (or *world* or *everybody*)
  : Same as *group*, but make the repository readable by all users.

  0xxx
  : *0xxx* is an octal number and each file will have mode *0xxx*. *0xxx* will override users' umask(2) value (and not only loosen permissions as *group* and *all* does). *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.

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

The template directory contains files and directories that 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 gitignore(5)), and sample hook files (see githooks(5)).

EXAMPLES

Start a new Git repository for an existing code base

```
$ cd /path/to/my/codebase
$ git init      <1>
$ git add .     <2>
$ git commit    <3>
```

1. Create a /path/to/my/codebase/.git directory.
2. Add all existing files to the index.
3. Record the pristine state as the first commit in the history.

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-init-db(1) Manual Page

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[=<permis

DESCRIPTION

This is a synonym for git-init(1). Please refer to the documentation of that command.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-instaweb(1) Manual Page

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 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 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 gitweb--browse(1) for more information about this.

SEE ALSO

gitweb(1)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-interpret-trailers(1) Manual Page

NAME

git-interpret-trailers - help add structured information into commit messages

SYNOPSIS

> *git interpret-trailers* [--trim-empty] [(--trailer <token>[(=|:)<value>])...] [<file>...]

DESCRIPTION

Help adding *trailers* lines, that look similar to RFC 822 e-mail headers, at the end of the otherwise free-form part of a commit message.

This command reads some patches or commit messages from either the <file> arguments or the standard input if no <file> is specified. Then this command applies the arguments passed using the `--trailer` option, if any, to the commit message part of each input file. The result is emitted on the standard output.

Some configuration variables control the way the `--trailer` arguments are applied to each commit message and the way any existing trailer in the commit message is changed. They also make it possible to automatically add some trailers.

By default, a *<token>=<value>* or *<token>:<value>* argument given using `--trailer` will be appended after the existing trailers only if the last trailer has a different (<token>, <value>) pair (or if there is no existing trailer). The <token> and <value> parts will be trimmed to remove starting and trailing whitespace, and the resulting trimmed <token> and <value> will appear in the message like this:

```
token: value
```

This means that the trimmed <token> and <value> will be separated by `': '` (one colon followed by one space).

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 after the commit message part of the output, and, if there is no line with only spaces at the end of the commit message part, one blank line will be added before the new trailer.

Existing trailers are extracted from the input message by looking for a group of one or more lines that contain a colon (by default), where the group is preceded by one or more empty (or whitespace-only) lines. The group must either be at the end of the message or be the last non-whitespace lines before a line that starts with ---. Such three minus signs start the patch part of the message.

When reading trailers, there can be whitespaces before and after the token, the separator and the value. There can also be whitespaces inside the token and the value.

Note that *trailers* do not follow and are not intended to follow many rules for RFC 822 headers. For example they do not follow the line folding rules, the encoding rules and probably many other rules.

OPTIONS

--trim-empty
: If the <value> part of any trailer contains only whitespace, the whole trailer will be removed from the resulting message. This apply to existing trailers as well as new trailers.

--trailer <token>[(=|:)<value>]
: Specify a (<token>, <value>) pair that should be applied as a trailer to the input messages. See the description of this command.

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 "%=$", then only lines using the format *<token><sep><value>* with <sep> containing *%*, = or *$* and then spaces will be considered trailers. And *%* will be the default separator used, so by default trailers will appear like: *<token>% <value>* (one percent sign and one space will appear between the token 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 <token>.

If it is `before`, then each new trailer will appear just before the first trailer with the same <token>.

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 <token> in the message.

The valid values for this option are: `addIfDifferentNeighbor` (this is the default), `addIfDifferent`, `add`, `overwrite` or `doNothing`.

With `addIfDifferentNeighbor`, a new trailer will be added only if no trailer with the same (<token>, <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 (<token>, <value>) pair is already in the message.

With `add`, a new trailer will be added, even if some trailers with the same (<token>, <value>) pair are already in the message.

With `replace`, an existing trailer with the same <token> will be deleted and the new trailer will be added. The deleted trailer will be the closest one (with the same <token>) 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 <token> in the message.

trailer.ifmissing
: This option makes it possible to choose what action will be performed when there is not yet any trailer with the same <token> in the message.

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.<token>.key
: This `key` will be used instead of <token> in the trailer. At the end of this 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, then the key will be used instead of both the <token> and the default separator when adding the trailer.

trailer.<token>.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 <token>.

trailer.<token>.ifexist
: This option takes the same values as the *trailer.ifexist* configuration variable and it overrides what is specified

by that option for trailers with the specified <token>.

trailer.<token>.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 <token>.

trailer.<token>.command
: This option can be used to specify a shell command that will be called to automatically add or modify a trailer with the specified <token>.

  When this option is specified, the behavior is as if a special *<token>=<value>* argument were added at the beginning of the command line, where <value> is taken to be the standard output of the specified command with any leading and trailing whitespace trimmed off.

  If the command contains the `$ARG` string, this string will be replaced with the <value> part of an existing trailer with the same <token>, if any, before the command is launched.

  If some *<token>=<value>* arguments are also passed on the command line, when a *trailer.<token>.command* is configured, the command will also be executed for each of these arguments. And the <value> part of these arguments, if any, will be used to replace the `$ARG` string in the command.

EXAMPLES

- Configure a *sign* trailer with a *Signed-off-by* key, and then add two of these trailers to a message:

```
$ git config trailer.sign.key "Signed-off-by"
$ cat msg.txt
subject

message
$ cat msg.txt | git interpret-trailers --trailer 'sign: Alice <alice@example.com>' --trailer 'sign: Bob
subject

message

Signed-off-by: Alice <alice@example.com>
Signed-off-by: Bob <bob@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@exa
```

- 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:

```
$ git config trailer.sign.key "Signed-off-by: "
$ git config trailer.sign.ifmissing add
$ git config trailer.sign.ifexists doNothing
$ git config trailer.sign.command 'echo "$(git config user.name) <$(git config user.email)>"'
$ git interpret-trailers <<EOF
> EOF

Signed-off-by: Bob <bob@example.com>
$ git interpret-trailers <<EOF
> Signed-off-by: Alice <alice@example.com>
> EOF

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 *see* trailer with a command to show the subject of a commit that is related, and show how it works:

```
$ git config trailer.see.key "See-also: "
$ git config trailer.see.ifExists "replace"
$ git config trailer.see.ifMissing "doNothing"
$ git config trailer.see.command "git log -1 --oneline --format=\"%h (%s)\" --abbrev-commit --abbrev=14
$ git interpret-trailers <<EOF
> subject
>
```

```
> message
>
> see: HEAD~2
> EOF
subject

message

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:

```
$ sed -e 's/ Z$/ /' >commit_template.txt <<EOF
> ***subject***
>
> ***message***
>
> Fixes: Z
> Cc: Z
> Reviewed-by: Z
> Signed-off-by: Z
> EOF
$ git config commit.template commit_template.txt
$ cat >.git/hooks/commit-msg <<EOF
> #!/bin/sh
> git interpret-trailers --trim-empty --trailer "git-version: \$(git describe)" "\$1" > "\$1.new"
> mv "\$1.new" "\$1"
> EOF
$ chmod +x .git/hooks/commit-msg
```

SEE ALSO

git-commit(1), git-format-patch(1), git-config(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-log(1) Manual Page

NAME

git-log - Show commit logs

SYNOPSIS

> *git log* [<options>] [<revision range>] [[--] <path>...]

DESCRIPTION

Shows the commit logs.

The command takes options applicable to the `git rev-list` command to control what is shown and how, and options applicable to the `git diff-*` commands 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|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. The default option is *short*.

--source
: Print out the ref name given on the command line by which each commit was reached.

--use-mailmap
: Use mailmap file to map author and committer names and email addresses to canonical real names and email addresses. See 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 :<regex>:<file>
: Trace the evolution of the line range given by "<start>,<end>" (or the funcname regex <regex>) 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. You can specify this option more than once.

  <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 ":<regex>" is given in place of <start> and <end>, it denotes the range from the first funcname line that matches <regex>, up to the next funcname line. ":<regex>" searches from the end of the previous `-L` range, if any, otherwise from the start of file. "^:<regex>" searches from the start of file.

<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 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.

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.

--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 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 `--show-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 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).

--perl-regexp
: Consider the limiting patterns to be Perl-compatible regular expressions. Requires libpcre to be compiled in.

--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
: 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. Cannot be combined with --bisect.

--not
: Reverses the meaning of the ^ prefix (or lack thereof) for all following revision specifiers, up to the next `--not`.

--all
: Pretend as if all the refs in `refs/` 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.

--reflog
: Pretend as if all objects mentioned by reflogs are listed on the command line as `<commit>`.

--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. Cannot be combined with --first-parent.

--stdin
: In addition to the *<commit>* listed on the command line, read them from the standard input. If a -- separator is seen, stop reading commits and start reading paths to limit the result.

--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 range, 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` (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. By default, *commit@{Nth}* notation is used in the output. When the starting commit is specified as *commit@{now}*, output also uses *commit@{timestamp}* notation instead. Under `--pretty=oneline`, the commit message is prefixed with this information on the same line. This option cannot be combined with `--reverse`. See also git-reflog(1).

--merge
: After a failed merge, show refs that touch files having a conflict and don't exist on all heads to merge.

--boundary
: Output excluded boundary commits. Boundary commits are prefixed with `-`.

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)

--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
: 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*.

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.

Finally, there is a fifth simplification mode available:

--ancestry-path
: Limit the displayed commits to those directly on the ancestry chain between the "from" and "to" commits in the given commit range. I.e. only display commits that are ancestor of the "to" commit and descendants of the "from" commit.

  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
```

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).

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 in reverse order. Cannot be combined with `--walk-reflogs`.

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`.

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*, *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 git-config(1)).

--abbrev-commit
: Instead of showing the full 40-byte hexadecimal commit object name, show only a partial prefix. Non default number of digits can be specified with "--abbrev=<n>" (which also modifies diff output, if it is displayed).

  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` and those options which imply it such as "--oneline". It also overrides the *log.abbrevCommit* variable.

--oneline
: This is a shorthand for "--pretty=oneline --abbrev-commit" used together.

--encoding=<encoding>
: The commit objects record the 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.

--notes[=<ref>]
: Show the notes (see 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 git-config(1) for more details.

  With an optional *<ref>* argument, show this notes ref instead of the default notes ref(s). The ref is taken to be in `refs/notes/` if it is not qualified.

  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[=<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=(relative|local|default|iso|iso-strict|rfc|short|raw)
: 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.

`--date=relative` shows dates relative to the current time, e.g. “2 hours ago”.

`--date=local` shows timestamps in user’s local time zone.

`--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 in the internal raw Git format `%s %z` format.

`--date=default` shows timestamps in the original time zone (either committer’s or author’s).

--parents
: Print also the parents of the commit (in the form "commit parent…"). Also enables parent rewriting, see *History Simplification* below.

--children
: Print also the children of the commit (in the form "commit child…"). Also enables parent rewriting, see *History Simplification* below.

--left-right
: Mark which side of a symmetric diff 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* below.

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.

Diff Formatting

Listed below are options that control the formatting of diff output. Some of them are specific to git-rev-list(1), however other diff options may be given. See git-diff-files(1) for more options.

-c
: With this option, diff output for a merge commit 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. Furthermore, it lists only files which were modified from all parents.

--cc
: This flag implies the `-c` option 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.

-m

This flag makes the merge commits show the full diff like regular commits; for each merge parent, a separate log entry and diff is generated. An exception is that only diff against the first parent is shown when `--first-parent` option is given; in that case, the output represents the changes the merge brought *into* the then-current branch.

-r

Show recursive diffs.

-t

Show the tree objects in the diff output. This implies `-r`.

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 sha1s 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 git-config(1)). Here are the details of the built-in formats:

- *oneline*

  ```
  <sha1> <title line>
  ```

  This is designed to be as compact as possible.

- *short*

  ```
  commit <sha1>
  Author: <author>

  <title line>
  ```

- *medium*

  ```
  commit <sha1>
  Author: <author>
  Date:   <author date>

  <title line>

  <full commit message>
  ```

- *full*

  ```
  commit <sha1>
  Author: <author>
  Commit: <committer>

  <title line>

  <full commit message>
  ```

- *fuller*

  ```
  commit <sha1>
  Author:     <author>
  AuthorDate: <author date>
  Commit:     <committer>
  CommitDate: <committer date>

  <title line>

  <full commit message>
  ```

- *email*

  ```
  From <sha1> <date>
  From: <author>
  Date: <author date>
  Subject: [PATCH] <title line>

  <full commit message>
  ```

- *raw*

The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the SHA-1s 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.

- *format:<string>*

  The *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:

  - *%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 git-shortlog(1) or git-blame(1))
  - *%ae*: author email
  - *%aE*: author email (respecting .mailmap, see git-shortlog(1) or 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
  - *%cn*: committer name
  - *%cN*: committer name (respecting .mailmap, see git-shortlog(1) or git-blame(1))
  - *%ce*: committer email
  - *%cE*: committer email (respecting .mailmap, see git-shortlog(1) or 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
  - *%d*: ref names, like the --decorate option of git-log(1)
  - *%D*: ref names without the " (", ")" wrapping.
  - *%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 signature, "B" for a Bad signature, "U" for a good, untrusted signature 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
  - *%gD*: reflog selector, e.g., `refs/stash@{1}`
  - *%gd*: shortened reflog selector, e.g., `stash@{1}`
  - *%gn*: reflog identity name

- *%gN*: reflog identity name (respecting .mailmap, see git-shortlog(1) or git-blame(1))
- *%ge*: reflog identity email
- *%gE*: reflog identity email (respecting .mailmap, see git-shortlog(1) or git-blame(1))
- *%gs*: reflog subject
- *%Cred*: switch color to red
- *%Cgreen*: switch color to green
- *%Cblue*: switch color to blue
- *%Creset*: reset color
- *%C(…)*: color specification, as described in color.branch.* config option; adding `auto,` at the beginning will emit color only when colors are 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). `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
- *%n*: newline
- *%%*: a raw *%*
- *%x00*: print a byte from a hex code
- *%w([<w>[,<i1>[,<i2>]]])*: switch line wrapping, like the -w option of git-shortlog(1).
- *%<(<N>[,trunc|ltrunc|mtrunc])*: make the next placeholder take at least N columns, padding spaces on the right if necessary. Optionally truncate at the beginning (ltrunc), the middle (mtrunc) or the end (trunc) if the output is longer than N columns. Note that truncating only works correctly with N >= 2.
- *%<|(<N>)*: make the next placeholder take at least until Nth columns, padding spaces on the right if necessary
- *%>(<N>)*, *%>|(<N>)*: similar to *%<(<N>)*, *%<|(<N>)* respectively, but padding spaces on the left
- *%>>(<N>)*, *%>>|(<N>)*: similar to *%>(<N>)*, *%>|(<N>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces
- *%><(<N>)*, *%><|(<N>)*: similar to *% <(<N>)*, *%<|(<N>)* respectively, but padding both sides (i.e. the text is centered)

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.

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, line-feeds that immediately precede 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
  ```

COMMON DIFF OPTIONS

-p

-u

--patch
: Generate patch (see section on generating patches).

-s

--no-patch
: Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>
: Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw
: Generate the raw format.

--patch-with-raw
: Synonym for `-p --raw`.

--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.

--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 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. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (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>`.

--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.

--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 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`.

--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 with new newlines.

  Also, when `--raw` or `--numstat` has been given, do not munge pathnames and use NULs as output field terminators.

  Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only
: Show only names of changed files.

--name-status
: Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]
: Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule(1) `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--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`.

--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.

  The regex can also be set via a diff driver or configuration option, see gitattributes(1) or 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.

--check
: Warn if changes introduce whitespace errors. What are considered whitespace 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. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--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 only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. 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 lack 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 require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--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.

-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.

-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 !regexec(regexp, two->ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

  While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

  See the *pickaxe* entry in gitdiffcore(7) for more information.

--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>
: Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config(1)). To cancel `diff.orderFile`, use `-O/dev/null`.

-R
: Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]
: 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.

-a

--text
: Treat all files as text.

--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.

--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.

-W

--function-context
: Show whole surrounding functions of changes.

--ext-diff
: Allow an external diff helper to be executed. If you set an external diff driver with gitattributes(5), you need to use this option with 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 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 git-diff(1) and git-log(1), but not for 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 git-config(1) or 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.

For more detailed explanation on these common options, see also gitdiffcore(7).

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with 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 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 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 SHA-1 checksum 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. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.
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
   ```

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 git-diff(1) or git-show(1). Note also that you can give the `-m' option to any of these commands to force generation of diffs with individual parents of a merge.

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 with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--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 contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.
3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

   Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.
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 for apply. 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

At the core level, Git is character encoding agnostic.

- The pathnames recorded in the index and in the tree objects are treated as uninterpreted sequences of non-NUL bytes. What readdir(2) returns are what are recorded and compared with the data Git keeps track of, which in turn are expected to be what lstat(2) and creat(2) accepts. There is no such thing as pathname encoding translation.
- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- The commit log messages are uninterpreted sequences of non-NUL bytes.

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* issues 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 its `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 git-config(1) for core variables and 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.

log.date
: Default format for human-readable dates. (Compare the `--date` option.) Defaults to "default", which means to write dates like `Sat May 8 19:35:34 2010 -0500`.

log.showRoot
: If `false`, `git log` and related commands will not treat the initial commit as a big creation event. Any root commits in `git log -p` output would be shown without a diff attached. The default is `true`.

mailmap.*
: See git-shortlog(1).

notes.displayRef
: Which refs, in addition to the default set by `core.notesRef` or *GIT_NOTES_REF*, to read notes from when showing commit messages with the `log` family of commands. See git-notes(1).

  May be an unabbreviated ref name or a glob and may be specified multiple times. 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, overridden by the *GIT_NOTES_DISPLAY_REF* environment variable, and overridden by the `--notes=<ref>` option.

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-ls-files(1) Manual Page

NAME

git-ls-files - Show information about files in the index and the working tree

SYNOPSIS

```
git ls-files [-z] [-t] [-v]
             (--[cached|deleted|others|ignored|stage|unmerged|killed|modified])*
             (-[c|d|o|i|s|u|k|m])*
             [-x <pattern>|--exclude=<pattern>]
             [-X <file>|--exclude-from=<file>]
             [--exclude-per-directory=<file>]
             [--exclude-standard]
             [--error-unmatch] [--with-tree=<tree-ish>]
             [--full-name] [--abbrev] [--] [<file>...]
```

DESCRIPTION

This merges the file listing in the directory cache index with the actual working directory list, and shows different combinations of the two.

One or more of the options below may be used to determine the files shown:

OPTIONS

-c

--cached
: Show cached files in the output (default)

-d

--deleted
: Show deleted files in the output

-m

--modified
: Show modified files in the output

-o

--others
: Show other (i.e. untracked) files in the output

-i

--ignored
: Show only ignored files in the output. When showing files in the index, print only those matched by an exclude pattern. When showing "other" files, show only those matched by an exclude pattern.

-s

--stage
: Show staged contents' object name, mode bits 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.

--no-empty-directory
: Do not list empty directories. Has no effect without --directory.

-u

--unmerged
: Show unmerged files in the output (forces --stage)

-k

--killed
: Show files on the filesystem that need to be removed due to file/directory conflicts for checkout-index to succeed.

-z
: \0 line termination on output.

-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>.

--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
: This feature is semi-deprecated. For scripting purpose, git-status(1) `--porcelain` and git-diff-files(1) `--name-status` are almost always superior alternatives, and users should look at git-status(1) `--short` or git-diff(1) `--name-status` for more user-friendly alternatives.

  This option identifies the file status with the following tags (followed by a space) at the start of each line:

H
: cached

S
: skip-worktree

M
: unmerged

R
: removed/deleted

C
: modified/changed

K
: to be killed

?
: other

-v
: Similar to `-t`, but use lowercase letters for files that are marked as *assume unchanged* (see 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.

--abbrev[=<n>]
: Instead of showing the full 40-byte hexadecimal object lines, show only a partial prefix. 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.

--
: 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 --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 git-read-tree(1) for more information on state)

When `-z` option is not used, TAB, LF, and backslash characters in pathnames are represented as `\t`, `\n`, and `\\`, respectively.

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. gitignore(5) specifies the format of exclude patterns.

These exclude patterns come from these 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.

SEE ALSO

git-read-tree(1), gitignore(5)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-ls-remote(1) Manual Page

NAME

git-ls-remote - List references in a remote repository

SYNOPSIS

> *git ls-remote* [--heads] [--tags] [-u <exec> | --upload-pack <exec>]
>   [--exit-code] <repository> [<refs>...]

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.

-u <exec>

--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 git-config(1)) and exit without talking to the remote.

<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 git-fetch(1)).

<refs>...
: When unspecified, all references, after filtering done with --heads and --tags, are shown. When <refs>... are specified, only references matching the given patterns are displayed.

EXAMPLES

```
$ git ls-remote --tags ./.
d6602ec5194c87b0fc87103ca4d67251c76f233a        refs/tags/v0.99
f25a265a342aed6041ab0cc484224d9ca54b6f41        refs/tags/v0.99.1
7ceca275d047c90c0c7d5afb13ab97efdf51bd6e        refs/tags/v0.99.3
c5db5456ae3b0873fc659c19fafdde22313cc441        refs/tags/v0.99.2
0918385dbd9656cab0d1d81ba7453d49bbc16250        refs/tags/junio-gpg-pub
$ git ls-remote http://www.kernel.org/pub/scm/git/git.git master pu rc
5fe978a5381f1fbad26a80e682ddd2a401966740        refs/heads/master
c781a84b5204fb294c9ccc79f8b3baceeb32c061        refs/heads/pu
$ git remote add korg http://www.kernel.org/pub/scm/git/git.git
$ git ls-remote --tags korg v\*
d6602ec5194c87b0fc87103ca4d67251c76f233a        refs/tags/v0.99
f25a265a342aed6041ab0cc484224d9ca54b6f41        refs/tags/v0.99.1
c5db5456ae3b0873fc659c19fafdde22313cc441        refs/tags/v0.99.2
7ceca275d047c90c0c7d5afb13ab97efdf51bd6e        refs/tags/v0.99.3
```

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-ls-tree(1) Manual Page

NAME

git-ls-tree - List the contents of a tree object

SYNOPSIS

> *git ls-tree* [-d] [-r] [-t] [-l] [-z]
> [--name-only] [--name-status] [--full-name] [--full-tree] [--abbrev[=<n>]]
> <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.

--name-only
--name-status
: List only filenames (instead of the "long" output), one per line.

--abbrev[=<n>]
: Instead of showing the full 40-byte hexadecimal object lines, show only a partial prefix. 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.

[<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

```
<mode> SP <type> SP <object> TAB <file>
```

Unless the `-z` option is used, TAB, LF, and backslash characters in pathnames are represented as `\t`, `\n`, and `\\`, respectively. This output format is compatible with what `--index-info --stdin` of *git update-index* expects.

When the `-l` option is used, format changes to

```
<mode> SP <type> SP <object> SP <object size> TAB <file>
```

Object size identified by <object> 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.

GIT

Part of the git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-mailinfo(1) Manual Page

NAME

git-mailinfo - Extracts patch and authorship from a single e-mail message

SYNOPSIS

> *git mailinfo* [-k|-b] [-u | --encoding=<encoding> | -n] [--[no-]scissors] <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 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. A line that mainly consists of scissors (either ">8" or "8<") and perforation (dash "-") marks is called a scissors line, and is used to request the reader to cut the message at that line. If such a 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 enabled by default with the configuration option mailinfo.scissors.

--no-scissors
: Ignore scissors lines. Useful for overriding mailinfo.scissors settings.

<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.

GIT

Part of the git(1) suite

Last updated 2014-12-23 17:41:32 CET

git-mailsplit(1) Manual Page

NAME

git-mailsplit - Simple UNIX mbox splitter program

SYNOPSIS

> *git mailsplit* [-b] [-f<nn>] [-d<prec>] [--keep-cr] -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 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`.

GIT

Part of the git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-merge(1) Manual Page

NAME

git-merge - Join two or more development histories together

SYNOPSIS

```
git merge [-n] [--stat] [--no-commit] [--squash] [--[no-]edit]
        [-s <strategy>] [-X <strategy-option>] [-S[<key-id>]]
        [--[no-]rerere-autoupdate] [-m <msg>] [<commit>...]
git merge <msg> HEAD <commit>...
git merge --abort
```

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.

```
          A---B---C topic
         /         \
    D---E---F---G---H master
```

The second syntax (<msg> `HEAD` <commit>...) is supported for historical reasons. Do not use it from the command line or in new scripts. It is the same as `git merge -m <msg> <commit>...`.

The third syntax ("`git merge --abort`") can only be run after the merge has resulted in conflicts. *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 but pretend the merge failed and do not autocommit, to give the user a chance to inspect and further tweak the merge result before committing.

--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.

--ff
: When the merge resolves as a fast-forward, only update the branch pointer, without creating a merge commit. This is the default behavior.

--no-ff
: Create a merge commit even when the merge resolves as a fast-forward. This is the default behaviour when merging an annotated (and possibly signed) tag.

--ff-only
: Refuse to merge and exit with a non-zero status unless the current `HEAD` is already up-to-date or the merge can be resolved as a fast-forward.

--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 git-fmt-merge-msg(1).

With --no-log do not list one-line descriptions from the actual commits being merged.

--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.

-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 (*git merge-recursive* when merging a single head, *git merge-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 commits being merged have good and trusted GPG signatures and abort the merge in case they do not.

--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.

-S[<keyid>]

--gpg-sign[=<keyid>]
: GPG-sign the resulting merge commit.

-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.

--[no-]rerere-autoupdate
: Allow the rerere mechanism to update the index with the result of auto-conflict resolution if possible.

--abort
: Abort the current conflict resolution process, and try to reconstruct the pre-merge state.

  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.

<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.

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 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. (One exception is when the changed index entries are in the state that would result from the merge already.)

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" program; i.e. 3-way merge results with familiar conflict markers `<<< === >>>`.
5. 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 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.
<<<<<<< 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 "diff3". 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
Conflict resolution is hard;
let's go shopping.
|||||||
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* to seal the deal.

You can work through the conflict with a number of tools:

- Use a mergetool. `git mergetool` to launch a graphical mergetool which will work you through the merge.
- Look at the diffs. `git diff` will show a three-way diff, highlighting changes from both the `HEAD` and `MERGE_HEAD` versions.
- 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`.

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 and is considered generally safe and fast.

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. This is the default merge strategy when pulling or merging one branch.

  The *recursive* 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 to 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*.

  patience
  : With this option, *merge-recursive* spends a little extra time to avoid mismerges that sometimes occur due to unimportant matching lines (e.g., braces from distinct functions). Use this when the branches to be merged have diverged wildly. See also git-diff(1) `--patience`.

  diff-algorithm=[patience|minimal|histogram|myers]
  : Tells *merge-recursive* to use a different diff algorithm, which can help avoid mismerges that occur due to unimportant matching lines (such as braces from distinct functions). See also git-diff(1) `--diff-algorithm`.

ignore-space-change

ignore-all-space

ignore-space-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 git-diff(1) `-b`, `-w`, and `--ignore-space-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 gitattributes(5) for details.

no-renormalize

Disables the `renormalize` option. This overrides the `merge.renormalize` configuration variable.

rename-threshold=<n>

Controls the similarity threshold used for rename detection. See also git-diff(1) `-M`.

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.

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 recursive 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, *recursive*), 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

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.

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.

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.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.renameLimit
: The number of files to consider when performing rename detection during a merge; if not specified, defaults to the value of diff.renameLimit.

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 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.tool
: Controls which merge tool is used by 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.

  - araxis
  - bc
  - bc3
  - codecompare
  - deltawalker
  - diffmerge
  - diffuse
  - ecmerge
  - emerge
  - gvimdiff
  - gvimdiff2
  - gvimdiff3
  - kdiff3
  - meld
  - opendiff
  - p4merge
  - tkdiff
  - tortoisemerge
  - vimdiff
  - vimdiff2
  - vimdiff3
  - xxdiff

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 gitattributes(5) for details.

merge.<driver>.driver
: Defines the command that implements a custom low-level merge driver. See 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 gitattributes(5) for details.

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.

SEE ALSO

git-fmt-merge-msg(1), git-pull(1), gitattributes(5), git-reset(1), git-diff(1), git-ls-files(1), git-add(1), git-rm(1), git-mergetool(1)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-merge-base(1) Manual Page

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 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

As 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 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 on 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 checkout -b 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---B1
                /
---o---o---B2--o---o---o---B (origin/master)
        \
         B3
          \
           Derived (topic)
```

where `origin/master` used to point at commits B3, B2, B1 and now it points at B, and your `topic` branch was started on top of it back when `origin/master` was at B3. This mode uses the reflog of `origin/master` to find B3 as the fork point, so that the `topic` can be rebased on top of the updated `origin/master` by:

```
$ fork_point=$(git merge-base --fork-point origin/master topic)
$ git rebase --onto origin/master $fork_point topic
```

See also

git-rev-list(1), git-show-branch(1), git-merge(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-merge-file(1) Manual Page

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] <current-file> <base-file> <other-file>

DESCRIPTION

git merge-file incorporates all changes that lead from the `<base-file>` to `<other-file>` into `<current-file>`. The result ordinarily goes into `<current-file>`. *git merge-file* is useful for combining separate changes to an original. Suppose `<base-file>` is the original, and both `<current-file>` and `<other-file>` are modifications of `<base-file>`, then *git merge-file* combines both changes.

A conflict occurs if both `<current-file>` and `<other-file>` 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-file>`, lines from `<other-file>`, or lines from both respectively. The length of the conflict markers can be given with the `--marker-size` option.

The exit value of this program is negative on error, and the number of conflicts otherwise. 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 git(1).

OPTIONS

-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-file>`.

-q
: Quiet; do not warn about conflicts.

--diff3
: Show conflicts in "diff3" style.

--ours

--theirs

--union
: Instead of leaving conflicts in the file, resolve conflicts favouring our (or their or both) side of the lines.

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

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-merge-index(1) Manual Page

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 git(1) suite

Last updated 2014-11-27 19:55:05 CET

git-merge-one-file(1) Manual Page

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 git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-mergetool(1) Manual Page

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 on 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.

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 as files are successfully merged.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-mergetool--lib(1) Manual Page

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.

get_merge_tool_cmd
: returns the custom command for a merge tool.

get_merge_tool_path
: returns the custom path for a merge tool.

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 git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-merge-tree(1) Manual Page

NAME

git-merge-tree - Show three-way merge without touching index

SYNOPSIS

> *git merge-tree* <base-tree> <branch1> <branch2>

DESCRIPTION

Reads three tree-ish, and output trivial merge results and conflicting stages to the standard output. This 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 is meant to be used by higher level scripts to compute merge results outside of the index, and stuff the results back into the index. For this reason, the output from the command omits entries that match the <branch1> tree.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-mktag(1) Manual Page

NAME

git-mktag - Creates a tag object

SYNOPSIS

> *git mktag* < signature_file

DESCRIPTION

Reads a tag contents on standard input and creates a tag object that can also be used to sign other objects.

The output is the new tag's <object> identifier.

Tag Format

A tag signature file has a very simple fixed format: four lines of

```
object <sha1>
type <typename>
tag <tagname>
tagger <tagger>
```

followed by some *optional* free-form message (some tags created by older Git may not have `tagger` line). The message, when 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 git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-mktree(1) Manual Page

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 normalised 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 sha1 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 as single blank line. The final new-line is optional. Note - if the *-z* option is used, lines are terminated with NUL.

GIT

Part of the git(1) suite

git-mv(1) Manual Page

NAME

git-mv - Move or rename a file, a directory, or a symlink

SYNOPSIS

> *git mv* <options>… <args>…

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 target 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 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 git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-name-rev(1) Manual Page

NAME

git-name-rev - Find symbolic names for given revs

SYNOPSIS

> *git name-rev* [--tags] [--refs=<pattern>]
> (--all | --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 one of branch name, tag name or fully qualified ref name.

--all
: List all commits reachable from all refs

--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. Intended for the scripter's use.

--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.

EXAMPLE

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 --stdin
```

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-notes(1) Manual Page

NAME

git-notes - Add or inspect object notes

SYNOPSIS

git notes [list [<object>]]
git notes add [-f] [--allow-empty] [-F <file> | -m <msg> | (-c | -C) <object>] [<object>]
git notes copy [-f] (--stdin | <from-object> <to-object>)
git notes append [--allow-empty] [-F <file> | -m <msg> | (-c | -C) <object>] [<object>]
git notes edit [--allow-empty] [<object>]
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" configuration in git-log(1).

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).

copy
: Copy the notes for the first object onto the second object. 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 to the notes of an existing object (defaults to HEAD). Creates a new notes object if needed.

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 -s/--strategy option) 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.

-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.

-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.)

-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.

--ref <ref>
: Manipulate the notes tree in <ref>. This overrides *GIT_NOTES_REF* and the "core.notesRef" configuration. The ref is taken to be in `refs/notes/` if it is not qualified.

--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 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". 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 a 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 [Permitted pathnames have the form *ab/cd/ef/.../abcdef...*: a sequence of directory names of two hexadecimal digits each followed by a filename with the rest of the object ID.]

.

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 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*.

"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 | 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.

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 on the command line or by the *GIT_NOTES_DISPLAY_REF* environment variable. See git-log(1).

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 by the *GIT_NOTES_REWRITE_REF* environment variable.

notes.rewriteMode
: When copying notes during a rewrite, what to do if the target commit already has a note. Must be one of `overwrite`, `concatenate`, and `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.

  Can be overridden with the *GIT_NOTES_REWRITE_REF* environment variable.

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`, and `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 git(7) suite

Last updated 2014-12-19 19:56:28 CET

git-p4(1) Manual Page

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.

EXAMPLE

- 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

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
```

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.

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
```

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
```

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*.

OPTIONS

General options

All commands except clone accept these options.

--git-dir <dir>
: Set the *GIT_DIR* environment variable. See git(1).

-v

--verbose
: Provide more progress information.

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 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>
: Limit the number of imported changes to *n*. Useful to limit the amount of history when using the *@all* p4 revision specifier.

--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.

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 git-clone(1).

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 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.

--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.

Rebase options

These options can be used to modify *git p4 rebase* behavior.

--import-labels
: Import p4 labels.

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 (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.

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.

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.

Submit variables

git-p4.detectRenames
: Detect renames. See git-diff(1). This can be true, false, or a score as expected by *git diff -M*.

git-p4.detectCopies
: Detect copies. See git-diff(1). This can be true, false, or a score as expected by *git diff -C*.

git-p4.detectCopiesHarder
: Detect copies harder. See 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*.

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.

Last updated 2015-03-26 21:44:44 CET

git-pack-objects(1) Manual Page

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]] [--stdout | base-name]
        [--shallow] [--keep-true-parents] < object-list
```

DESCRIPTION

Reads list of objects from the standard input, and writes a packed archive with specified base-name, or 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 a pair of files (.pack and .idx), using <base-name> to determine the name of the created file. When this option is used, the two files 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.

--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.

--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, which is the default.

--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. The default is unlimited, unless the config variable `pack.packSizeLimit` is set.

--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.

--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.

--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 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 per-cent.

  Note: Porcelain commands such as `git gc` (see git-gc(1)), `git repack` (see 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 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.

SEE ALSO

git-rev-list(1) git-repack(1) git-prune-packed(1)

GIT

Part of the git(1) suite

Last updated 2015-01-25 20:40:15 CET

git-pack-redundant(1) Manual Page

NAME

git-pack-redundant - Find redundant pack files

SYNOPSIS

> *git pack-redundant* [--verbose] [--alt-odb] < --all | .pack filename ... >

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 directories to be present in local packs.

--verbose
: Outputs some statistics to stderr. Has a small performance penalty.

SEE ALSO

git-pack-objects(1) git-repack(1) git-prune-packed(1)

GIT

Part of the git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-pack-refs(1) Manual Page

NAME

git-pack-refs - Pack heads and tags for efficient repository access

SYNOPSIS

> *git pack-refs* [--all] [--no-prune]

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 branch tips to be packed as well. 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.

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 git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-parse-remote(1) Manual Page

NAME

git-parse-remote - Routines to help parsing remote repository access parameters

SYNOPSIS

> *. "$(git --exec-path)/git-parse-remote"*

DESCRIPTION

This script is included in various scripts to supply routines to parse files under $GIT_DIR/remotes/ and

$GIT_DIR/branches/ and configuration variables that are related to fetching, pulling and pushing.

GIT

Part of the git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-patch-id(1) Manual Page

NAME

git-patch-id - Compute unique ID for a patch

SYNOPSIS

> *git patch-id* [--stable | --unstable] < <patch>

DESCRIPTION

A "patch ID" is nothing but a sum of SHA-1 of the file diffs associated with a patch, with whitespace and 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.

IOW, you can use this thing 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

--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.

  ```
  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. 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.
  ```

<patch>
: The diff to create the ID of.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-prune(1) Manual Page

NAME

git-prune - Prune all unreachable objects from the object database

SYNOPSIS

> *git prune* [-n] [-v] [--expire <expire>] [--] [<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 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 git-repack(1).

OPTIONS

-n

--dry-run

Do not remove anything; just report what it would remove.

-v

--verbose

Report all removed objects.

--

Do not interpret any more arguments as options.

--expire <time>

Only expire loose objects older than <time>.

<head>...

In addition to objects reachable from any of our references, keep objects reachable from listed <head>s.

EXAMPLE

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

git-fsck(1), git-gc(1), git-reflog(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-prune-packed(1) Manual Page

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 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

git-pack-objects(1) git-repack(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-pull(1) Manual Page

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. In its default mode, `git pull` is shorthand for `git fetch` followed by `git merge FETCH_HEAD`.

More precisely, *git pull* runs *git fetch* with the given parameters and calls *git merge* to merge the retrieved branch heads into the current branch. With `--rebase`, it runs *git rebase* instead of *git merge*.

<repository> should be the name of a remote repository as passed to 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 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 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 cancelled and the work tree untouched. It is generally best to get any local changes in working order before pulling or stash them away with git-stash(1).

OPTIONS

Options meant for *git pull* itself and the underlying *git merge* must be given before the options meant for *git fetch*.

-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 all populated submodules should be fetched too (see git-config(1) and gitmodules(5)). That might be necessary to get the data needed for merging submodule commits, a feature Git learned in 1.7.3. Notice that the result of a merge will not be checked out in the submodule, "git submodule update" has to be called afterwards to bring the work tree up to date with the merge result.

Options related to merging

--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 but pretend the merge failed and do not autocommit, to give the user a chance to inspect and further tweak the merge result before committing.

--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.

--ff
: When the merge resolves as a fast-forward, only update the branch pointer, without creating a merge commit. This is the default behavior.

--no-ff
: Create a merge commit even when the merge resolves as a fast-forward. This is the default behaviour when merging an annotated (and possibly signed) tag.

--ff-only
: Refuse to merge and exit with a non-zero status unless the current `HEAD` is already up-to-date or the merge can be resolved as a fast-forward.

--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 git-fmt-merge-msg(1).

  With --no-log do not list one-line descriptions from the actual commits being merged.

--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.

-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 (*git merge-recursive* when merging a single head, *git merge-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 commits being merged have good and trusted GPG signatures and abort the merge in case they do not.

--summary

--no-summary
: Synonyms to --stat and --no-stat; these are deprecated and will be removed in the future.

-r

--rebase[=false|true|preserve]
: 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 preserve, rebase with the `--preserve-merges` option passed to `git rebase` so that locally created merge commits will not be flattened.

  When false, merge the current branch into the upstream branch.

  See `pull.rebase`, `branch.<name>.rebase` and `branch.autoSetupRebase` in 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 git-rebase(1) carefully.

--no-rebase
: Override earlier --rebase.

Options related to fetching

--all
: Fetch all remotes.

-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.

--depth=<depth>
: Deepen or shorten the history of a *shallow* repository created by `git clone` with `--depth=<depth>` option (see git-clone(1)) to the specified number of commits from the tip of each remote branch history. Tags for the deepened commits are not fetched.

--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 accept such refs.

-f

--force
: When *git fetch* is used with `<rbranch>:<lbranch>` refspec, it refuses to update the local branch `<lbranch>` unless the remote branch `<rbranch>` it fetches is a descendant of `<lbranch>`. This option overrides that check.

-k

--keep
: Keep downloaded pack.

--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 git-config(1).

-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.

--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.

<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 git-fetch(1)).

  The format of a <refspec> parameter is an optional plus `+`, followed by the source ref <src>, followed by a colon `:`, followed by the destination ref <dst>. The colon can be omitted when <dst> is empty.

  `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 empty string, the local ref that matches it is fast-forwarded using <src>. If the optional plus `+` is used, the local ref is updated even if it does not result in a fast-forward update.

  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 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 and rsync can be used for fetching and pushing, but these are 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.xz[:port]/path/to/repo.git/
- git://host.xz[:port]/path/to/repo.git/
- http[s]://host.xz[:port]/path/to/repo.git/
- ftp[s]://host.xz[:port]/path/to/repo.git/
- rsync://host.xz/path/to/repo.git/

An alternative scp-like syntax may also be used with the ssh protocol:

- [user@]host.xz:path/to/repo.git/

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.xz[:port]/~[user]/path/to/repo.git/
- git://host.xz[:port]/~[user]/path/to/repo.git/
- [user@]host.xz:/~[user]/path/to/repo.git/

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 git-clone(1) for details.

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 gitremote-helpers(1) 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.

Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using git-remote(1), 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>`.

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 format
        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.

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`.

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 and is considered generally safe and fast.

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. This is the default merge strategy when pulling or merging one branch.

  The *recursive* 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 to 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*.

patience
: With this option, *merge-recursive* spends a little extra time to avoid mismerges that sometimes occur due to unimportant matching lines (e.g., braces from distinct functions). Use this when the branches to be merged have diverged wildly. See also git-diff(1) `--patience`.

diff-algorithm=[patience|minimal|histogram|myers]
: Tells *merge-recursive* to use a different diff algorithm, which can help avoid mismerges that occur due to unimportant matching lines (such as braces from distinct functions). See also git-diff(1) `--diff-algorithm`.

ignore-space-change

ignore-all-space

ignore-space-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 git-diff(1) `-b`, `-w`, and `--ignore-space-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 gitattributes(5) for details.

no-renormalize
: Disables the `renormalize` option. This overrides the `merge.renormalize` configuration variable.

rename-threshold=<n>
: Controls the similarity threshold used for rename detection. See also git-diff(1) `-M`.

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.

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 recursive 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, *recursive*), 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 `` URL: ` line in `$GIT_DIR/remotes/<origin> `` file 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>` file 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 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, but does not update any remote-tracking branches. Using remote-tracking branches, 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*.

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 can not 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

git-fetch(1), git-merge(1), git-config(1)

GIT

Part of the git(1) suite

Last updated 2015-05-03 21:16:24 CEST

git-push(1) Manual Page

NAME

git-push - Update remote refs along with associated objects

SYNOPSIS

```
git push [--all | --mirror | --tags] [--follow-tags] [--atomic] [-n | --dry-run] [--receive-pack=<git-receive-pack>]
         [--repo=<repository>] [-f | --force] [--prune] [-v | --verbose]
         [-u | --set-upstream] [--signed]
         [--force-with-lease[=<refname>[:<expect>]]]
         [--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 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 git-config(1) for the meaning of `push.default`).

OPTIONS

<repository>
: The "remote" repository that is 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 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>`.

The object referenced by <src> is used to update the <dst> reference on the remote side. By default this is only allowed if <dst> is not a tag (annotated or lightweight), and then only if it can fast-forward <dst>. By having the optional leading `+`, you can tell Git to update the <dst> ref even if it is not allowed by default (e.g., it is not a fast-forward.) This does **not** attempt to merge <src> into <dst>. See EXAMPLES below for details.

`tag <tag>` means the same as `refs/tags/<tag>:refs/tags/<tag>`.

Pushing an empty <src> allows you to delete the <dst> ref from the remote repository.

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.

--all
: 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.

--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 git-config(1).

--signed
: 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. See 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.

--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 her commit, and blindly pushing with `--force` will lose her 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).

  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.

-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.

--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 git-pull(1) and other commands. For more information, see *branch.<name>.merge* in git-config(1).

--[no-]thin
: These options are passed to 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.

--recurse-submodules=check|on-demand
: 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.

--[no-]verify
: Toggle the pre-push hook (see githooks(5)). The default is --verify, giving the hook a chance to prevent the push. With --no-verify, the hook is bypassed completely.

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 and rsync can be used for fetching and pushing, but these are 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.xz[:port]/path/to/repo.git/
- git://host.xz[:port]/path/to/repo.git/
- http[s]://host.xz[:port]/path/to/repo.git/
- ftp[s]://host.xz[:port]/path/to/repo.git/
- rsync://host.xz/path/to/repo.git/

An alternative scp-like syntax may also be used with the ssh protocol:

- [user@]host.xz:path/to/repo.git/

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.xz[:port]/~[user]/path/to/repo.git/
- git://host.xz[:port]/~[user]/path/to/repo.git/
- [user@]host.xz:/~[user]/path/to/repo.git/

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 git-clone(1) for details.

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 gitremote-helpers(1) 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.

Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using git-remote(1), 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>`.

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 format
        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.

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 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 (`remote.origin.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`.

  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.

GIT

Part of the git(1) suite

Last updated 2015-05-03 21:16:24 CEST

git-quiltimport(1) Manual Page

NAME

git-quiltimport - Applies a quilt patchset onto the current branch

SYNOPSIS

> *git quiltimport* [--dry-run | -n] [--author <author>] [--patches <dir>]

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 and the quilt series file.

The default for the patch directory is patches or the value of the $QUILT_PATCHES environment variable.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-read-tree(1) Manual Page

NAME

git-read-tree - Reads tree information into the index

SYNOPSIS

```
git read-tree [[-m [--trivial] [--aggressive] | --reset | --prefix=<prefix>]
                [-u [--exclude-per-directory=<gitignore>] | -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: 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.

Trivial merges are done by *git read-tree* itself. Only conflicting paths will be in 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 previous merge you started.

--reset
: Same as -m, except that unmerged entries are discarded instead of failing.

-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. Note that the

`<prefix>/` value must end with a slash.

--exclude-per-directory=<gitignore>
: When running the command with `-u` and `-m` options, the merge result may need to overwrite paths that are not tracked in the current branch. The command usually refuses to proceed with the merge to avoid losing such a path. However this safety valve sometimes gets in the way. For example, it often happens that the other branch added a file that used to be a generated file in your branch, and the safety valve triggers when you try to switch to that branch after you ran `make` but before running `make clean` to remove the generated file. This option tells the command to read per-directory exclude file (usually *.gitignore*) and allows such an untracked but explicitly ignored file to be overwritten.

--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-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.

<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 trees are provided.

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*.

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-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

"Sparse checkout" allows populating the working directory sparsely. It uses the skip-worktree bit (see 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, 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

git-write-tree(1); git-ls-files(1); gitignore(5)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-rebase(1) Manual Page

NAME

git-rebase - Forward-port local commits to the updated upstream head

SYNOPSIS

> *git rebase* [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
> [<upstream> [<branch>]]
> *git rebase* [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
> --root [<branch>]
> *git rebase* --continue | --skip | --abort | --edit-todo

DESCRIPTION

If <branch> is specified, *git rebase* will perform an automatic `git checkout <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 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.

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. 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
```

CONFIGURATION

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 *--autosquash* option by default.

rebase.autoStash
: If set to true enable *--autostash* option by default.

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.

<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.

--continue
: Restart the rebasing process after having resolved a merge conflict.

--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.

--keep-empty
: Keep the commits that do not change anything from its parents in the result.

--skip
: Restart the rebasing process by skipping the current patch.

--edit-todo
: Edit the todo list during an interactive rebase.

-m

--merge

Use merging strategies to rebase. When the recursive (default) merge strategy is used, this allows rebase to be aware of renames on the upstream side.

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.

-s <strategy>

--strategy=<strategy>
: Use the given merge strategy. If there is no `-s` option *git merge-recursive* is used instead. 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 discards all patches from the <branch>, which makes little sense.

-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 recursive`. Note the reversal of *ours* and *theirs* as noted above for the `-m` option.

-S[<keyid>]

--gpg-sign[=<keyid>]
: GPG-sign commits.

-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 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 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.

-f

--force-rebase
: Force a rebase even if the current branch is up-to-date and the command without `--force` would return without doing anything.

You may find this (or --no-ff with an interactive rebase) 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 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 git-merge-base(1)). If *fork_point* ends up being empty, the <upstream> will be used as a fallback.

If either <upstream> or --root is given on the command line, then the default is `--no-fork-point`, otherwise the default is `--fork-point`.

--ignore-whitespace

--whitespace=<option>
: These flag are passed to the *git apply* program (see git-apply(1)) that applies the patch. Incompatible with the --interactive option.

--committer-date-is-author-date

--ignore-date
: These flags are passed to *git am* to easily change the dates of the rebased commits (see git-am(1)). Incompatible

with the --interactive option.

-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).

-p

--preserve-merges
: Recreate merge commits instead of flattening the history by replaying commits a merge commit introduces. Merge conflict resolutions or manual amendments to merge commits are not preserved.

  This uses the `--interactive` machinery internally, but combining it with the `--interactive` option explicitly is generally not a good idea unless you know what you are doing (see BUGS 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.

  This option can only be used with the `--interactive` option (see INTERACTIVE MODE below).

  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.

--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. When used with --onto, it will skip changes already contained in <newbase> (instead of <upstream>) whereas without --onto it will operate on every change. When used together with both --onto and --preserve-merges, *all* root commits will be rewritten to have <newbase> as parent instead.

--autosquash

--no-autosquash
: When the commit log message begins with "squash! …" (or "fixup! …"), and there is a commit whose title begins with the same …, automatically modify the todo list of rebase -i so that the commit marked for squashing comes right after the commit to be modified, and change the action of the moved commit from `pick` to `squash` (or `fixup`). Ignores subsequent "fixup! " or "squash! " after the first, in case you referred to an earlier fixup/squash with `git commit --fixup/--squash`.

  This option is only valid when the *--interactive* option is used.

  If the *--autosquash* option is enabled by default using the configuration variable `rebase.autoSquash`, this option can be used to override and disable this setting.

--[no-]autostash
: Automatically create a temporary stash 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.

--no-ff
: With --interactive, cherry-pick 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.

  Without --interactive, this is a synonym for --force-rebase.

  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 for details).

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`.

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 and is considered generally safe and fast.

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. This is the default merge strategy when pulling or merging one branch.

  The *recursive* 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 to 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*.

  patience
  : With this option, *merge-recursive* spends a little extra time to avoid mismerges that sometimes occur due to unimportant matching lines (e.g., braces from distinct functions). Use this when the branches to be merged have diverged wildly. See also git-diff(1) `--patience`.

  diff-algorithm=[patience|minimal|histogram|myers]
  : Tells *merge-recursive* to use a different diff algorithm, which can help avoid mismerges that occur due to unimportant matching lines (such as braces from distinct functions). See also git-diff(1) `--diff-algorithm`.

  ignore-space-change

  ignore-all-space

  ignore-space-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 git-diff(1) `-b`, `-w`, and `--ignore-space-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 gitattributes(5) for details.

  no-renormalize
  : Disables the `renormalize` option. This overrides the `merge.renormalize` configuration variable.

  rename-threshold=<n>
  : Controls the similarity threshold used for rename detection. See also git-diff(1) `-M`.

  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.

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 recursive 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, *recursive*), 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 git-rebase command 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.

If you just want to edit the commit message for a commit, replace the command "pick" with the command "reword".

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 commit messages of the first commit and of those with the "squash" command, but omits the commit messages of commits with the "fixup" command.

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 preserve merges, 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 -p --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 one specified in `$SHELL`, or the default shell if `$SHELL` is not set), 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---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 `filter-branch`.

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. 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'--o'  subsystem
                                                   \
                                                    *---*---*  topic
```

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 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!

BUGS

The todo list presented by `--preserve-merges --interactive` does not represent the topology of the revision graph. Editing commits and rewording their commit messages should work fine, but attempts to reorder commits tend to produce counterintuitive results.

For example, an attempt to rearrange

```
1 --- 2 --- 3 --- 4 --- 5
```

to

```
1 --- 2 --- 4 --- 3 --- 5
```

by moving the "pick 4" line will result in the following history:

```
        3
       /
1 --- 2 --- 4 --- 5
```

GIT

Part of the git(1) suite

Last updated 2015-05-03 21:16:24 CEST

git-receive-pack(1) Manual Page

NAME

git-receive-pack - Receive what is pushed into the repository

SYNOPSIS

git-receive-pack <directory>

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 remote repository. For pull operations, see git-fetch-pack(1).

The command allows for 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.

OPTIONS

<directory>
    The repository to sync into.

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 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 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 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.

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
```

SEE ALSO

git-send-pack(1), gitnamespaces(7)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-reflog(1) Manual Page

NAME

git-reflog - Manage reflog information

SYNOPSIS

> *git reflog* <subcommand> <options>

DESCRIPTION

The command takes various subcommands, and different options depending on the subcommand:

> *git reflog* [*show*] [log-options] [<ref>]
> *git reflog expire* [--expire=<time>] [--expire-unreachable=<time>]
>   [--rewrite] [--updateref] [--stale-fix]
>   [--dry-run] [--verbose] [--all | <refs>...]
> *git reflog delete* [--rewrite] [--updateref]
>   [--dry-run] [--verbose] ref@{specifier}...

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 gitrevisions(7) for more details.

This command manages the information recorded in the reflogs.

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 git-log(1) for more information.

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 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.

OPTIONS

Options for `show`

`git reflog show` accepts any of the options accepted by `git log`.

Options for `expire`

--all
: Process the reflogs of all references.

--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.

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 git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-relink(1) Manual Page

NAME

git-relink - Hardlink common objects in local repositories

SYNOPSIS

> *git relink* [--safe] <dir>... <master_dir>

DESCRIPTION

This will scan 1 or more object repositories and look for objects in common with a master repository. Objects not already hardlinked to the master repository will be replaced with a hardlink to the master repository.

OPTIONS

--safe
: Stops if two objects with the same hash exist but have different sizes. Default is to warn and continue.

<dir>
: Directories containing a .git/objects/ subdirectory.

GIT

Part of the git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-remote(1) Manual Page

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* <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 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. 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
: Adds 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 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.

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 all stale remote-tracking branches under <name>. These stale branches have already been removed from the remote repository referenced by <name>, but are still locally available in "remotes/<name>".

With `--dry-run` option, report what branches will be pruned, but do not actually prune them.

update

Fetch updates for a named set of remotes in the repository as defined by remotes.<group>. If a named group is not 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 git-config(1)).

With `--prune` option, prune all the remotes that are updated.

DISCUSSION

The remote configuration is achieved using the `remote.origin.url` and `remote.origin.fetch` configuration variables. (See git-config(1)).

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 checkout -b 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

git-fetch(1) git-branch(1) git-config(1)

GIT

Part of the git(1) suite

git-remote-ext(1) Manual Page

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 rest of the argument. Default is not to send such a request.

  This is useful if 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 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 user on remote side. This avoids needing to edit .ssh/config.

"ext::socat -t3600 - ABSTRACT-CONNECT:/git-server %G/somerepo"
: Represents repository with path /somerepo accessible over git protocol at 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

gitremote-helpers(1)

GIT

Part of the git(1) suite

Last updated 2014-12-13 19:39:10 CET

git-remote-fd(1) Manual Page

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 remote Git server (git-upload-pack, git-receive-pack or git-upload-achive). 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 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 same service.

`git push fd::7,8/bar master`
: Same as above.

SEE ALSO

gitremote-helpers(1)

GIT

Part of the git(1) suite

Last updated 2014-12-13 19:39:10 CET

git-remote-helpers

This document has been moved to gitremote-helpers(1).

Please let the owners of the referring site know so that they can update the link you clicked to get here.

Thanks.

Last updated 2014-11-27 19:54:14 CET

git-remote-testgit(1) Manual Page

NAME

git-remote-testgit - Example remote-helper

SYNOPSIS

> git clone testgit::<source-repo> [<destination>]

DESCRIPTION

This command is a simple remote-helper, that is used both as a testcase for the remote-helper functionality, and as an example to show remote-helper authors one possible implementation.

The best way to learn more is to read the comments and source code in *git-remote-testgit*.

SEE ALSO

gitremote-helpers(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-repack(1) Manual Page

NAME

git-repack - Pack unpacked objects in a repository

SYNOPSIS

> *git repack* [-a] [-A] [-d] [-f] [-F] [-l] [-n] [-q] [-b] [--window=<n>] [--depth=<n>]

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.

-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 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.

-l
: Pass the `--local` option to *git pack-objects*. See git-pack-objects(1).

-f
: Pass the `--no-reuse-delta` option to `git-pack-objects`, see git-pack-objects(1).

-F
: Pass the `--no-reuse-object` option to `git-pack-objects`, see git-pack-objects(1).

-q
: Pass the `-q` option to *git pack-objects*. See 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 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.

--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, which is the default.

--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. The default is unlimited, unless the config variable `pack.packSizeLimit` is set.

-b

--write-bitmap-index
: Write a reachability bitmap index as part of the repack. This only makes sense when used with `-a` or `-A`, as the bitmaps must be able to refer to all reachable objects. This option overrides the setting of `pack.writeBitmaps`.

--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 `pack.writeBitmaps`, as it ensures that the bitmapped packfile has the necessary objects.

Configuration

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 or rsync 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.

SEE ALSO

git-pack-objects(1) git-prune-packed(1)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-replace(1) Manual Page

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* -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 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 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. See contrib/convert-grafts-to-replace-refs.sh for an example script based on this option that can convert grafts to replace refs.

-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 format are available:

- *short*: <replaced sha1>
- *medium*: <replaced sha1> → <replacement sha1>
- *long*: <replaced sha1> (<replaced type>) → <replacement sha1> (<replacement type>)

CREATING REPLACEMENT OBJECTS

git-filter-branch(1), git-hash-object(1) and git-rebase(1), 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

git-hash-object(1) git-filter-branch(1) git-rebase(1) git-tag(1) git-branch(1) git-commit(1) git-var(1) git(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-request-pull(1) Manual Page

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, 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.

EXAMPLE

Imagine that you built your work on your `master` branch on top of the `v1.0` release, and want it to be integrated to 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 git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-rerere(1) Manual Page

NAME

git-rerere - Reuse recorded resolution of conflicted merges

SYNOPSIS

> *git rerere* [*clear*|*forget* <pathspec>|*diff*|*remaining*|*status*|*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 checkout 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 checkout topic
        $ git merge master
        $ ... work on both topic and master branches
        $ git checkout 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 checkout topic
        $ git merge master
        $ git reset --hard HEAD^ ;# rewind the test merge
        $ ... work on both topic and master branches
        $ git checkout 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.

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-reset(1) Manual Page

NAME

git-reset - Reset current HEAD to the specified state

SYNOPSIS

> *git reset* [-q] [<tree-ish>] [--] <paths>...
> *git reset* (--patch | -p) [<tree-ish>] [--] [<paths>...]
> *git reset* [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]

DESCRIPTION

In the first and second form, copy entries from <tree-ish> to the index. In the third 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>] [--] <paths>...
: This form resets the index entries for all <paths> to their state at <tree-ish>. (It does not affect the working tree or the current branch.)

  This means that `git reset <paths>` is the opposite of `git add <paths>`.

  After running `git reset <paths>` to update the index entry, you can use git-checkout(1) to check the contents out of the index to the working tree. Alternatively, using git-checkout(1) and specifying a commit, 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>] [--] [<paths>...]
: 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 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>. 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 git-add(1)).

  --hard
  : Resets the index and working tree. Any changes to tracked files in the working tree since <commit> are discarded.

  --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.

If you want to undo a commit other than the latest on a branch, git-revert(1) is your friend.

OPTIONS

-q

--quiet
: Be quiet, only report errors.

EXAMPLES

Undo add

```
$ edit                                     <1>
$ git add frotz.c filfre.c
$ mailx                                    <2>
$ git reset                                <3>
$ git pull git://info.example.com/ nitfol  <4>
```

1. 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.
2. Somebody asks you to pull, and the changes sounds worthy of merging.
3. 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.
4. 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^      <1>
$ edit                        <2>
$ git commit -a -c ORIG_HEAD  <3>
```

1. 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".
2. Make corrections to working tree files.
3. "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 git-commit(1).

Undo a commit, making it a topic branch

```
$ git branch topic/wip     <1>
$ git reset --hard HEAD~3  <2>
$ git checkout topic/wip   <3>
```

1. 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.
2. Rewind the master branch to get rid of those three commits.
3. Switch to "topic/wip" branch and keep working.

Undo commits permanently

```
$ git commit ...
$ git reset --hard HEAD~3   <1>
```

1. 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 git-rebase(1) for the implications of doing so.)

Undo a merge or pull

```
$ git pull                         <1>
Auto-merging nitfol
CONFLICT (content): Merge conflict in nitfol
Automatic merge failed; fix conflicts and then commit the result.
$ git reset --hard                 <2>
$ git pull . topic/branch          <3>
Updating from 41223... to 13134...
Fast-forward
$ git reset --hard ORIG_HEAD       <4>
```

1. 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.
2. "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.
3. Merge a topic branch into the current branch, which resulted in a fast-forward.
4. 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                         <1>
Auto-merging nitfol
Merge made by recursive.
 nitfol                |   20 +++++----
 ...
$ git reset --merge ORIG_HEAD      <2>
```

1. 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.
2. 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 checkout feature ;# you were working in "feature" branch and
$ work work work       ;# got interrupted
$ git commit -a -m "snapshot WIP"                 <1>
$ git checkout master
$ fix fix fix
$ git commit ;# commit with real log
$ git checkout feature
$ git reset --soft HEAD^ ;# go back to WIP state  <2>
$ git reset                                       <3>
```

1. This commit will get blown away so a throw-away log message is OK.
2. This removes the *WIP* commit from the commit history, and sets your working tree to the state just before you made that snapshot.
3. 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 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                      <1>
$ git commit -m "Commit files in index"     <2>
$ git add frotz.c                           <3>
```

1. This removes the file from the index while keeping it in the working directory.
2. This commits all other changes in the index.
3. 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 checkout -b branch1
$ edit
$ git commit ...                            <1>
$ edit
$ git checkout -b branch2                   <2>
$ git reset --keep start                    <3>
```

1. This commits your first edits in branch1.
2. 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 checkout -b branch2 start"), but nobody is perfect.
3. But you can use "reset --keep" to remove the unwanted commit after you switched to "branch2".

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 local change wrt 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 git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-revert(1) Manual Page

NAME

git-revert - Revert some existing commits

SYNOPSIS

git revert [--[no-]edit] [-n] [-m parent-number] [-s] [-S[<key-id>]] <commit>...
git revert --continue
git revert --quit
git revert --abort

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 git-reset(1), particularly the *--hard* option. If you want to extract specific files as they were in another commit, you should see git-checkout(1), specifically the `git checkout <commit> -- <filename>` syntax. Take care with these alternatives as both will discard uncommitted changes in your working directory.

OPTIONS

<commit>...
: Commits to revert. For a more complete list of ways to spell commit names, see gitrevisions(7). Sets of commits can also be given but no traversal is done by default, see 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 for more details.

--no-edit
: With this option, *git revert* will not start the commit message editor.

-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[<key-id>]

--gpg-sign[=<key-id>]
: GPG-sign commits.

-s

--signoff
: Add Signed-off-by line at the end of the commit message.

--strategy=<strategy>
: Use the given merge strategy. Should only be used once. See the MERGE STRATEGIES section in git-merge(1) for details.

-X<option>

--strategy-option=<option>
: Pass the merge strategy-specific option through to the merge strategy. See git-merge(1) for details.

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.

--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.

SEE ALSO

git-cherry-pick(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-rev-list(1) Manual Page

NAME

git-rev-list - Lists commit objects in reverse chronological order

SYNOPSIS

```
git rev-list [ --max-count=<number> ]
             [ --skip=<number> ]
             [ --max-age=<timestamp> ]
             [ --min-age=<timestamp> ]
             [ --sparse ]
             [ --merges ]
             [ --no-merges ]
             [ --min-parents=<number> ]
             [ --no-min-parents ]
             [ --max-parents=<number> ]
             [ --no-max-parents ]
             [ --first-parent ]
             [ --remove-empty ]
             [ --full-history ]
             [ --not ]
             [ --all ]
             [ --branches[=<pattern>] ]
             [ --tags[=<pattern>] ]
             [ --remotes[=<pattern>] ]
             [ --glob=<glob-pattern> ]
             [ --ignore-missing ]
             [ --stdin ]
             [ --quiet ]
             [ --topo-order ]
             [ --parents ]
             [ --timestamp ]
             [ --left-right ]
             [ --left-only ]
             [ --right-only ]
             [ --cherry-mark ]
             [ --cherry-pick ]
             [ --encoding=<encoding> ]
             [ --(author|committer|grep)=<pattern> ]
             [ --regexp-ignore-case | -i ]
             [ --extended-regexp | -E ]
             [ --fixed-strings | -F ]
             [ --date=(local|relative|default|iso|iso-strict|rfc|short) ]
             [ [ --objects | --objects-edge | --objects-edge-aggressive ]
               [ --unpacked ] ]
[ --pretty | --header ]
[ --bisect ]
[ --bisect-vars ]
[ --bisect-all ]
[ --merge ]
[ --reverse ]
[ --walk-reflogs ]
[ --no-walk ] [ --do-walk ]
[ --use-bitmap-index ]
<commit>... [ -- <paths>... ]
```

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 given on the command line form a set of commits that are reachable from any of them, 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 a very 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 enables it to be used by commands as different as *git bisect* and *git repack*.

OPTIONS

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.

--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 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 `--show-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 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).

--perl-regexp
: Consider the limiting patterns to be Perl-compatible regular expressions. Requires libpcre to be compiled in.

--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
: 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. Cannot be combined with --bisect.

--not
: Reverses the meaning of the *^* prefix (or lack thereof) for all following revision specifiers, up to the next `--not`.

--all
: Pretend as if all the refs in `refs/` 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.

--reflog
: Pretend as if all objects mentioned by reflogs are listed on the command line as `<commit>`.

--ignore-missing
: Upon seeing an invalid object name in the input, pretend as if the bad input was not given.

--stdin
: In addition to the *<commit>* listed on the command line, read them from the standard input. If a -- separator is seen, stop reading commits and start reading paths to limit the result.

--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.

--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 range, 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` (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. By default, *commit@{Nth}* notation is used in the output. When the starting commit is specified as *commit@{now}*, output also uses *commit@{timestamp}* notation instead. Under `--pretty=oneline`, the commit message is prefixed with this information on the same line. This option cannot be combined with `--reverse`. See also git-reflog(1).

--merge
: After a failed merge, show refs that touch files having a conflict and don't exist on all heads to 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.

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)

--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
: 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*.

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.

Finally, there is a fifth simplification mode available:

--ancestry-path
: Limit the displayed commits to those directly on the ancestry chain between the “from” and “to” commits in the given commit range. I.e. only display commits that are ancestor of the “to” commit and descendants of the “from” commit.

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
```

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).

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. Cannot be combined with --first-parent.

--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.

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 in reverse order. Cannot be combined with `--walk-reflogs`.

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*”.

--objects-edge
: Similar to `--objects`, but also print the IDs of excluded commits prefixed with a “-” character. This is used by 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.

--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`.

Commit Formatting

Using these options, git-rev-list(1) will act similar to the more specialized family of commit log tools: git-log(1), git-show(1), and 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*, *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 git-config(1)).

--abbrev-commit
: Instead of showing the full 40-byte hexadecimal commit object name, show only a partial prefix. Non default number of digits can be specified with "--abbrev=<n>" (which also modifies diff output, if it is displayed).

  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` and those options which imply it such as "--oneline". It also overrides the *log.abbrevCommit* variable.

--oneline
: This is a shorthand for "--pretty=oneline --abbrev-commit" used together.

--encoding=<encoding>
: The commit objects record the 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.

--notes[=<ref>]
: Show the notes (see 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 git-config(1) for more details.

  With an optional *<ref>* argument, show this notes ref instead of the default notes ref(s). The ref is taken to be in `refs/notes/` if it is not qualified.

  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[=<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=(relative|local|default|iso|iso-strict|rfc|short|raw)
: 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.

  `--date=relative` shows dates relative to the current time, e.g. "2 hours ago".

  `--date=local` shows timestamps in user's local time zone.

  `--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 in the internal raw Git format `%s %z` format.

`--date=default` shows timestamps in the original time zone (either committer's or author's).

--header
: Print the contents of the commit in raw-format; each record is separated with a NUL character.

--parents
: Print also the parents of the commit (in the form "commit parent…"). Also enables parent rewriting, see *History Simplification* below.

--children
: Print also the children of the commit (in the form "commit child…"). Also enables parent rewriting, see *History Simplification* below.

--timestamp
: Print the raw commit timestamp.

--left-right
: Mark which side of a symmetric diff 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* below.

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 sha1s 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 git-config(1)). Here are the details of the built-in formats:

- *oneline*

  ```
  <sha1> <title line>
  ```

  This is designed to be as compact as possible.

- *short*

```
commit <sha1>
Author: <author>

<title line>
```

- *medium*

  ```
  commit <sha1>
  Author: <author>
  Date:   <author date>

  <title line>

  <full commit message>
  ```

- *full*

  ```
  commit <sha1>
  Author: <author>
  Commit: <committer>

  <title line>

  <full commit message>
  ```

- *fuller*

  ```
  commit <sha1>
  Author:     <author>
  AuthorDate: <author date>
  Commit:     <committer>
  CommitDate: <committer date>

  <title line>

  <full commit message>
  ```

- *email*

  ```
  From <sha1> <date>
  From: <author>
  Date: <author date>
  Subject: [PATCH] <title line>

  <full commit message>
  ```

- *raw*

  The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the SHA-1s 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.

- *format:<string>*

  The *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:

  - *%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 git-shortlog(1) or git-blame(1))
  - *%ae*: author email
  - *%aE*: author email (respecting .mailmap, see git-shortlog(1) or 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
- *%cn*: committer name
- *%cN*: committer name (respecting .mailmap, see git-shortlog(1) or git-blame(1))
- *%ce*: committer email
- *%cE*: committer email (respecting .mailmap, see git-shortlog(1) or 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
- *%d*: ref names, like the --decorate option of git-log(1)
- *%D*: ref names without the " (", ")" wrapping.
- *%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 signature, "B" for a Bad signature, "U" for a good, untrusted signature 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
- *%gD*: reflog selector, e.g., `refs/stash@{1}`
- *%gd*: shortened reflog selector, e.g., `stash@{1}`
- *%gn*: reflog identity name
- *%gN*: reflog identity name (respecting .mailmap, see git-shortlog(1) or git-blame(1))
- *%ge*: reflog identity email
- *%gE*: reflog identity email (respecting .mailmap, see git-shortlog(1) or git-blame(1))
- *%gs*: reflog subject
- *%Cred*: switch color to red
- *%Cgreen*: switch color to green
- *%Cblue*: switch color to blue
- *%Creset*: reset color
- *%C(...)*: color specification, as described in color.branch.* config option; adding `auto,` at the beginning will emit color only when colors are 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). `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
- *%n*: newline
- *%%*: a raw *%*
- *%x00*: print a byte from a hex code
- *%w([<w>[,<i1>[,<i2>]]])*: switch line wrapping, like the -w option of git-shortlog(1).
- *%<(<N>[,trunc|ltrunc|mtrunc])*: make the next placeholder take at least N columns, padding spaces on the right if necessary. Optionally truncate at the beginning (ltrunc), the middle (mtrunc) or the end (trunc) if the output is longer than N columns. Note that truncating only works correctly with N >= 2.
- *%<|(<N>)*: make the next placeholder take at least until Nth columns, padding spaces on the right if necessary

- *%>(<N>)*, *%>|(<N>)*: similar to *%<(<N>)*, *%<|(<N>)* respectively, but padding spaces on the left
- *%>>(<N>)*, *%>>|(<N>)*: similar to *%>(<N>)*, *%>|(<N>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces
- *%><(<N>)*, *%><|(<N>)*: similar to *% <(<N>)*, *%<|(<N>)* respectively, but padding both sides (i.e. the text is centered)

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.

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, line-feeds that immediately precede 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
```

GIT

Part of the git(1) suite

Last updated 2015-01-25 20:40:15 CET

git-rev-parse(1) Manual Page

NAME

git-rev-parse - Pick out and massage parameters

SYNOPSIS

> *git rev-parse* [--option] <args>...

DESCRIPTION

Many Git porcelainish commands take 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*. This command is used to distinguish between them.

OPTIONS

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).

--sq-quote
: Use *git rev-parse* in shell quoting mode (see SQ-QUOTE section below). In contrast to the `--sq` option below, this mode does only quoting. Nothing else is done to command input.

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.

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.

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)"
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 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.

-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.

--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.

--short

--short=number
: Instead of outputting the full SHA-1 values of object names try to abbreviate them to a shorter unique name. When no length is specified 7 is used. The minimum length is 4.

--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 show them as full refnames (e.g. "refs/heads/master").

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.

--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.

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.

--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.

--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".

--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.

--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-toplevel
: Show the absolute path of the top-level directory.

--shared-index-path
: Show the path to the shared index file in split index mode, or empty if not in split-index mode.

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*.

<args>...
: 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.

<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* and *CHERRY_PICK_HEAD*);
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, 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`. *CHERRY_PICK_HEAD* records the commit which you are cherry-picking when you run `git cherry-pick`.

   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}*
: The suffix *@{upstream}* to a branchname (short form *<branchname>@{u}*) refers to the branch that the branch specified by branchname is set to build on top of (configured with `branch.<name>.remote` and `branch.<name>.merge`). A missing branchname defaults to the current one.

*<rev>^*, 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. *master~3*
: 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. If the commit message starts with a *!* you have to repeat that; the special sequence *:/!*, followed by something else than *!*, is reserved for now. The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. *:/^foo*.

<rev>:<path>, e.g. *HEAD:README*, *: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. *:path* (with an empty part before the colon) is a special case of the syntax described next: content recorded in the index at the given path. 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  = 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. To these commands, specifying a single revision with the notation described in the previous section means the set of commits reachable from that commit, following the commit ancestry chain.

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*.

This 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*.

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* or *r2* but not from both.

In these two shorthands, 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.

Two other shorthands for naming a set that is formed by a commit and its parent commits exist. The *r1^@* notation means all parents of *r1*. *r1^!* includes commit *r1* but excludes all of its parents.

To summarize:

<rev>
: Include commits that are reachable from (i.e. ancestors of) <rev>.

^<rev>
: Exclude commits that are reachable from (i.e. ancestors of) <rev>.

<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 then all its parents prefixed with *^* to exclude them (and their ancestors).

Here are a handful of examples:

```
D                 G H D
D F               G H I J D F
^G D              H D
^D B              E I J F B
B..C              C
```

```
B...C              G H D E B C
^D B C             E I J F B C
C                  I J F C
C^@                I J F
C^!                C
F^! 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.

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. `h,help`, `dry-run` and `f` are all three 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 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 to 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).

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 $?)"
```

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
    --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
```

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.

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 $REV^{commit}
  ```

  This will error out if $REV is empty or not a valid revision.
- Similar to above:

  ```
  $ git rev-parse --default master --verify $REV
  ```

  but if $REV is empty, the commit object name from master will be printed.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-rm(1) Manual Page

NAME

git-rm - Remove files from the working tree and from the index

SYNOPSIS

> *git rm* [-f | --force] [-n] [-r] [--cached] [--ignore-unmatch] [--quiet] [--] <file>…

DESCRIPTION

Remove files 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.

OPTIONS

<file>…
: Files to remove. Fileglobs (e.g. `*.c`) can be given to remove all matching files. If you want Git to expand file glob characters, you may need to shell-escape them. 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.

-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.

-q

--quiet
: `git rm` normally outputs one line (in the form of an `rm` command) for each file removed. This option suppresses that output.

DISCUSSION

The <file> list given to the command can be exact pathnames, file glob patterns, or leading directory names. The command removes only the paths that are known to Git. Giving the name of a file that you have not told Git about does not remove that file.

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`.

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.

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`.

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 git-add(1).

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 fail - no matter if forced or not - to protect the submodule's history. If it exists the submodule.<name> section in the 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 submodules 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 git-submodule(1) `deinit` instead.

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

git-add(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-send-email(1) Manual Page

NAME

git-send-email - Send a collection of patches as emails

SYNOPSIS

> *git send-email* [options] <file|directory|rev-list options>…

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 git-format-patch(1) can be passed to git send-email.

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 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

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*.

  The --bcc option must be repeated for each user you want on the bcc list.

--cc=<address>
: Specify a starting "Cc:" value for each email. Default is the value of *sendemail.cc*.

  The --cc option must be repeated for each user you want on the cc list.

--compose
: Invoke a text editor (see GIT_EDITOR in git-var(1)) to edit an introductory message for the patch series.

  When *--compose* is used, git send-email will use the From, Subject, 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 From, Subject, and In-Reply-To headers 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".

--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.

The --to option must be repeated for each user you want on the to list.

--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)
: 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. Default is the value of the *sendemail.transferEncoding* configuration value; if that is unspecified, git will use 8bit and not add a Content-Transfer-Encoding header.

--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`.

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.

--smtp-encryption=<encryption>
: Specify the encryption to use, either *ssl* or *tls*. Any other value reverts to plain SMTP. 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-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*.

--smtp-server=<host>
: If set, specifies the outgoing SMTP server to use (e.g. `smtp.example.com` or a raw IP address). Alternatively it can specify a full pathname of a sendmail-like program instead; the program must support the `-i` option. Default value can be specified by the *sendemail.smtpServer* configuration option; the built-in default is `/usr/sbin/sendmail` or `/usr/lib/sendmail` if such program is available, or `localhost` otherwise.

--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.

Automating

--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.

--[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 Signed-off-by: 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 Signed-off-by lines except for self (use *self* for that).
- *cccmd* will avoid running the --cc-cmd.
- *body* is equivalent to *sob* + *bodycc*
- *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.

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:

- Warn of patches that contain lines longer than 998 characters; this is due to SMTP limits as described by http://www.ietf.org/rfc/rfc2821.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.

CONFIGURATION

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*, or *gnus*.

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 previous section for the meaning of these values.

EXAMPLE

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
```

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/*
```

Note: the following perl modules are required Net::SMTP::SSL, MIME::Base64 and Authen::SASL

SEE ALSO

git-format-patch(1), git-imap-send(1), mbox(5)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-send-pack(1) Manual Page

NAME

git-send-pack - Push objects over Git protocol to another repository

SYNOPSIS

> *git send-pack* [--all] [--dry-run] [--force] [--receive-pack=<git-receive-pack>] [--verbose] [--thin] [--atomic] [<host>:

DESCRIPTION

Usually you would want to use *git push*, which is a higher-level wrapper of this command, instead. See 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. What this means is 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.

<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 *--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 pattern 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 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 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 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 in order to avoid accidentally overwriting the remote ref and lose 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 git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-shell(1) Manual Page

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 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.

EXAMPLE

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
```

SEE ALSO

ssh(1), git-daemon(1), contrib/git-shell-commands/README

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-sh-i18n(1) Manual Page

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 git-sh-i18n--envsubst(1) helper. Will be replaced by a real gettext implementation in a later version.

GIT

Part of the git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-sh-i18n--envsubst(1) Manual Page

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 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 git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-shortlog(1) Manual Page

NAME

git-shortlog - Summarize 'git log' output

SYNOPSIS

```
git log --pretty=short | git shortlog [<options>]
git shortlog [<options>] [<revision range>] [[--] <path>...]
```

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 git-log(1).)

```
Each pretty-printed commit will be rewrapped before it is shown.
```

-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 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.

MAPPING AUTHORS

The `.mailmap` feature is used to coalesce together commits by the same person in the shortlog, where their name and/or email address was spelled differently.

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, it is used to map author and committer names and email addresses to canonical real names and email addresses.

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.

Example 1: 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 proper `.mailmap` file would look like:

```
Jane Doe         <jane@desktop.(none)>
Joe R. Developer <joe@example.com>
```

Note how there is no need for an entry for `<jane@laptop.(none)>`, because the real name of that author is already correct.

Example 2: Your repository contains commits from the following authors:

```
nick1 <bugs@company.xx>
nick2 <bugs@company.xx>
nick2 <nick2@company.xx>
santa <me@company.xx>
claus <me@company.xx>
CTO <cto@coompany.xx>
```

Then you might want a `.mailmap` file that looks like:

```
<cto@company.xx>                       <cto@coompany.xx>
Some Dude <some@dude.xx>         nick1 <bugs@company.xx>
Other Author <other@author.xx>   nick2 <bugs@company.xx>
Other Author <other@author.xx>         <nick2@company.xx>
Santa Claus <santa.claus@northpole.xx> <me@company.xx>
```

Use hash # for comments that are either on their own line, or after the email address.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:55:05 CET

git-show(1) Manual Page

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.

The command takes options applicable to the *git diff-tree* command 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. For a more complete list of ways to spell object names, see "SPECIFYING REVISIONS" section in 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*, *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 git-config(1)).

--abbrev-commit
: Instead of showing the full 40-byte hexadecimal commit object name, show only a partial prefix. Non default number of digits can be specified with "--abbrev=<n>" (which also modifies diff output, if it is displayed).

  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` and those options which imply it such as "--oneline". It also overrides the *log.abbrevCommit* variable.

--oneline
: This is a shorthand for "--pretty=oneline --abbrev-commit" used together.

--encoding=<encoding>
: The commit objects record the 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.

--notes[=<ref>]
: Show the notes (see 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 git-config(1) for more details.

  With an optional *<ref>* argument, show this notes ref instead of the default notes ref(s). The ref is taken to be in `refs/notes/` if it is not qualified.

  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[=<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 sha1s 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 git-config(1)). Here are the details of the built-in formats:

- *oneline*

  ```
  <sha1> <title line>
  ```

  This is designed to be as compact as possible.

- *short*

  ```
  commit <sha1>
  Author: <author>

  <title line>
  ```

- *medium*

  ```
  commit <sha1>
  Author: <author>
  Date:   <author date>

  <title line>

  <full commit message>
  ```

- *full*

  ```
  commit <sha1>
  Author: <author>
  Commit: <committer>

  <title line>

  <full commit message>
  ```

- *fuller*

  ```
  commit <sha1>
  Author:     <author>
  AuthorDate: <author date>
  Commit:     <committer>
  CommitDate: <committer date>

  <title line>

  <full commit message>
  ```

- *email*

  ```
  From <sha1> <date>
  From: <author>
  Date: <author date>
  Subject: [PATCH] <title line>

  <full commit message>
  ```

- *raw*

  The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the SHA-1s 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.

- *format:<string>*

  The *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:

- *%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 git-shortlog(1) or git-blame(1))
- *%ae*: author email
- *%aE*: author email (respecting .mailmap, see git-shortlog(1) or 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
- *%cn*: committer name
- *%cN*: committer name (respecting .mailmap, see git-shortlog(1) or git-blame(1))
- *%ce*: committer email
- *%cE*: committer email (respecting .mailmap, see git-shortlog(1) or 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
- *%d*: ref names, like the --decorate option of git-log(1)
- *%D*: ref names without the " (", ")" wrapping.
- *%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 signature, "B" for a Bad signature, "U" for a good, untrusted signature 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
- *%gD*: reflog selector, e.g., `refs/stash@{1}`
- *%gd*: shortened reflog selector, e.g., `stash@{1}`
- *%gn*: reflog identity name
- *%gN*: reflog identity name (respecting .mailmap, see git-shortlog(1) or git-blame(1))
- *%ge*: reflog identity email
- *%gE*: reflog identity email (respecting .mailmap, see git-shortlog(1) or git-blame(1))
- *%gs*: reflog subject
- *%Cred*: switch color to red
- *%Cgreen*: switch color to green
- *%Cblue*: switch color to blue
- *%Creset*: reset color

- *%C(...)*: color specification, as described in color.branch.* config option; adding `auto,` at the beginning will emit color only when colors are 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). `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
- *%n*: newline
- *%%*: a raw *%*
- *%x00*: print a byte from a hex code
- *%w([<w>[,<i1>[,<i2>]]])*: switch line wrapping, like the -w option of git-shortlog(1).
- *%<(<N>[,trunc|ltrunc|mtrunc])*: make the next placeholder take at least N columns, padding spaces on the right if necessary. Optionally truncate at the beginning (ltrunc), the middle (mtrunc) or the end (trunc) if the output is longer than N columns. Note that truncating only works correctly with N >= 2.
- *%<|(<N>)*: make the next placeholder take at least until Nth columns, padding spaces on the right if necessary
- *%>(<N>)*, *%>|(<N>)*: similar to *%<(<N>)*, *%<|(<N>)* respectively, but padding spaces on the left
- *%>>(<N>)*, *%>>|(<N>)*: similar to *%>(<N>)*, *%>|(<N>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces
- *%><(<N>)*, *%><|(<N>)*: similar to *% <(<N>)*, *%<|(<N>)* respectively, but padding both sides (i.e. the text is centered)

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.

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, line-feeds that immediately precede 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
```

COMMON DIFF OPTIONS

-p

-u

--patch
: Generate patch (see section on generating patches).

-s

--no-patch
: Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>
: Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw
: Generate the raw format.

--patch-with-raw
: Synonym for `-p --raw`.

--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.

--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 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. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (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>`.

--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.

--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 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`.

--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 with new newlines.

  Also, when `--raw` or `--numstat` has been given, do not munge pathnames and use NULs as output field terminators.

  Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only
: Show only names of changed files.

--name-status
: Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]
: Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule(1) `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--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`.

--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.

  The regex can also be set via a diff driver or configuration option, see gitattributes(1) or 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.

--check
: Warn if changes introduce whitespace errors. What are considered whitespace 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. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--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 only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. 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 lack 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 require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--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.

-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.

-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 !regexec(regexp, two->ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

  While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

  See the *pickaxe* entry in gitdiffcore(7) for more information.

--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>
: Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config(1)). To cancel `diff.orderFile`, use `-O/dev/null`.

-R
: Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]
: 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.

-a

--text
: Treat all files as text.

--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.

--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.

-W

--function-context
: Show whole surrounding functions of changes.

--ext-diff
: Allow an external diff helper to be executed. If you set an external diff driver with gitattributes(5), you need to use this option with 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 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 git-diff(1) and git-log(1), but not for 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 git-config(1) or 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.

For more detailed explanation on these common options, see also gitdiffcore(7).

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with 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 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 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 SHA-1 checksum 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. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.

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
   ```

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 git-diff(1) or git-show(1). Note also that you can give the `-m' option to any of these commands to force generation of diffs with individual parents of a merge.

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 with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--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 contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

   Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.

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 for apply. 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 tags 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

At the core level, Git is character encoding agnostic.

- The pathnames recorded in the index and in the tree objects are treated as uninterpreted sequences of non-NUL bytes. What readdir(2) returns are what are recorded and compared with the data Git keeps track of, which in turn are expected to be what lstat(2) and creat(2) accepts. There is no such thing as pathname encoding translation.
- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- The commit log messages are uninterpreted sequences of non-NUL bytes.

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* issues 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 its `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 git(1) suite

git-show-branch(1) Manual Page

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 <globs>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 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 to 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 <reference>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 git-merge-base(1) handles the case of three or more commits.

--independent
: Among the <reference>s given, display only the ones that cannot be reached from any other <reference>.

--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 <references>, 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, 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 heads.
 +  [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".

EXAMPLE

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 with each other.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-show-index(1) Manual Page

NAME

git-show-index - Show packed archive index

SYNOPSIS

> *git show-index* < idx-file

DESCRIPTION

Reads given idx file for packed Git archive created with *git pack-objects* command, and dumps its contents.

The information it outputs is subset of what you can get from *git verify-pack -v*; this command only shows the packfile offset and SHA-1 of each object.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:55:05 CET

git-show-ref(1) Manual Page

NAME

git-show-ref - List references in a local repository

SYNOPSIS

> *git show-ref* [-q|--quiet] [--verify] [--head] [-d|--dereference]

> [-s|--hash[=<n>]] [--abbrev[=<n>]] [--tags]
> [--heads] [--] [<pattern>...]
> *git show-ref* --exclude-existing[=<pattern>] < ref-list

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 shows the refs from stdin that don't exist in the local repository.

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.

--tags

--heads
: 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 SHA-1 hash, not the reference name. When combined with --dereference the dereferenced tag will still be shown after the SHA-1.

--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.

--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. When combined with *--verify* this can be used 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: *<SHA-1 ID> <space> <reference name>*.

```
$ 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 format is: *<SHA-1 ID>*

```
$ git show-ref --heads --hash
2e3ba0114a1f52b47df29743d6915d056be13278
185008ae97960c8d551adcd9e23565194651b5d1
03adf42c988195b50e1a1935ba5fcbc39b2b029b
...
```

EXAMPLE

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 exists.

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

git-for-each-ref(1), git-ls-remote(1), git-update-ref(1), gitrepository-layout(5)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-sh-setup(1) Manual Page

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 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 git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-stage(1) Manual Page

NAME

git-stage - Add file contents to the staging area

SYNOPSIS

```
git stage args…
```

DESCRIPTION

This is a synonym for git-add(1). Please refer to the documentation of that command.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-stash(1) Manual Page

NAME

git-stash - Stash the changes in a dirty working directory away

SYNOPSIS

```
git stash list [<options>]
git stash show [<stash>]
git stash drop [-q|--quiet] [<stash>]
git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>]
git stash branch <branchname> [<stash>]
git stash [save [-p|--patch] [-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 save`. 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).

OPTIONS

save [-p|--patch] [-k|--[no-]keep-index] [-u|--include-untracked] [-a|--all] [-q|--quiet] [<message>]

Save your local modifications to a new *stash*, and run `git reset --hard` to revert them. The <message> part is optional and gives the description along with the stashed state. For quickly making a snapshot, you can omit *both* "save" and <message>, but giving only <message> does not trigger this action to prevent a misspelled subcommand from making an unwanted stash.

If the `--keep-index` option is used, all changes already added to the index are left intact.

If the `--include-untracked` option is used, all untracked files are also stashed and then cleaned up with `git clean`, leaving the working directory in a very clean state. If the `--all` option is used instead then the ignored files are stashed and cleaned in addition to the untracked files.

With `--patch`, you can 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 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.

list [<options>]

List the stashes that you currently have. Each *stash* is listed with its name (e.g. `stash@{0}` is the latest stash, `stash@{1}` is the one before, etc.), the name of the branch that was current when the stash was made, and a short description of the commit the stash 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 git-log(1).

show [<stash>]

Show the changes recorded in the stash as a diff between the stashed state and its original parent. When no `<stash>` is given, shows the latest one. 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 stash in patch form).

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 save`. 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.

If the `--index` option is used, then 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).

When no `<stash>` is given, `stash@{0}` is assumed, otherwise `<stash>` must be a reference of the form `stash@{<revision>}`.

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 save` 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>`. When no `<stash>` is given, applies the latest one.

This is useful if the branch on which you ran `git stash save` has changed enough that `git stash apply` fails due to conflicts. Since the stash 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 stashed states. Note that those states 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 stashed state from the stash list. When no `<stash>` is given, it removes the latest one. i.e. `stash@{0}`, otherwise `<stash>` must be a valid stash log reference of the form `stash@{<revision>}`.

create
: Create a stash (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 "save" 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 "save" above.

DISCUSSION

A stash 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 stash was created. The tree of the second parent records the state of the index when the stash 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 checkout -b my_wip
$ git commit -a -m "WIP"
$ git checkout master
$ edit emergency fix
$ git commit -a -m "Fix in a hurry"
$ git checkout 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 save --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 save --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'
```

Recovering stashes that were cleared/dropped erroneously

If you mistakenly drop or clear stashes, they cannot be recovered through the normal safety mechanisms. However, you can try the following incantation to get a list of stashes 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
```

SEE ALSO

git-checkout(1), git-commit(1), git-reflog(1), git-reset(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-status(1) Manual Page

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 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.

--porcelain

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.

--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 optional (defaults to *all*), and is used to specify the handling of untracked files.

  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. You can use `no` to have `git status` return more quickly without showing untracked files.

    The default can be changed using the status.showUntrackedFiles configuration variable documented in 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 git-config(1) or 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
: Show ignored files as well.

-z
: Terminate entries with NUL, instead of LF. This implies the `--porcelain` 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.

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.

Short Format

In the short-format, the status of each path is shown as

```
XY PATH1 -> PATH2
```

where `PATH1` is the path in the `HEAD`, and the "` -> PATH2`" part is shown only when `PATH1` corresponds to a different path in the index/worktree (i.e. the file is renamed). 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.

For paths with merge conflicts, `X` and `Y` show the modification states of each side of the merge. For paths that do not have merge conflicts, `X` shows the status of the index, and `Y` shows the status of the work tree. For untracked paths, `XY` are `??`. Other status codes can be interpreted as follows:

- ' ' = unmodified
- *M* = modified
- *A* = added
- *D* = deleted
- *R* = renamed
- *C* = copied
- *U* = updated but unmerged

Ignored files are not listed, unless `--ignored` option is in effect, in which case `XY` are `!!`.

```
X          Y     Meaning
-------------------------------------------------
          [MD]   not updated
M        [ MD]   updated in index
A        [ MD]   added to index
D         [ M]   deleted from index
R        [ MD]   renamed in index
C        [ MD]   copied in index
[MARC]           index and work tree matches
[ MARC]      M   work tree changed since index
[ MARC]      D   deleted 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
-------------------------------------------------
```

If -b is used the short-format status is preceded by a line

## branchname tracking info

Porcelain Format

The 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.

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 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.

SEE ALSO

gitignore(5)

GIT

Part of the git(1) suite

Last updated 2015-05-03 21:16:24 CEST

git-stripspace(1) Manual Page

NAME

git-stripspace - Remove unnecessary whitespace

SYNOPSIS

> *git stripspace* [-s | --strip-comments] < input
> *git stripspace* [-c | --comment-lines] < input

DESCRIPTION

Clean the input in the manner used by Git for text such as commit messages, notes, tags and branch descriptions.

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 git-apply(1) for correcting whitespace of patches or files in the repository.

OPTIONS

-s

--strip-comments
  Skip and remove all lines starting with comment character (default *#*).

-c

--comment-lines
  Prepend comment character and blank 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 git(1) suite

Last updated 2014-12-23 17:41:32 CET

git-submodule(1) Manual Page

NAME

git-submodule - Initialize, update or inspect submodules

SYNOPSIS

```
git submodule [--quiet] add [-b <branch>] [-f|--force] [--name <name>]
              [--reference <repository>] [--depth <depth>] [--] <repository> [<path>]
git submodule [--quiet] status [--cached] [--recursive] [--] [<path>...]
git submodule [--quiet] init [--] [<path>...]
git submodule [--quiet] deinit [-f|--force] [--] <path>...
git submodule [--quiet] update [--init] [--remote] [-N|--no-fetch]
              [-f|--force] [--rebase|--merge] [--reference <repository>]
              [--depth <depth>] [--recursive] [--] [<path>...]
git submodule [--quiet] summary [--cached|--files] [(-n|--summary-limit) <n>]
              [commit] [--] [<path>...]
git submodule [--quiet] foreach [--recursive] <command>
git submodule [--quiet] sync [--recursive] [--] [<path>...]
```

DESCRIPTION

Submodules allow foreign repositories to be embedded within a dedicated subdirectory of the source tree, always pointed at a particular commit.

They are not to be confused with remotes, which are meant mainly for branches of the same project; submodules are meant for different projects you would like to make part of your source tree, while the history of the two projects still stays completely independent and you cannot modify the contents of the submodule from within the main project. If you want to merge the project histories and want to treat the aggregated whole as a single project from then on, you may want to add a remote for the other project and use the *subtree* merge strategy, instead of treating the other project as a submodule. Directories that come from both projects can be cloned and checked out as a whole if you choose to go that route.

Submodules are composed from a so-called `gitlink` tree entry in the main repository that refers to a particular commit object within the inner repository that is completely separate. A record in the `.gitmodules` (see gitmodules(5)) file at the root of the source tree assigns a logical name to the submodule and describes the default URL the submodule shall be cloned from. The logical name can be used for overriding this URL within your local repository configuration (see *submodule init*).

This command will manage the tree entries and contents of the gitmodules file for you, as well as inspect the status of your submodules and update them. When adding a new submodule to the tree, the *add* subcommand is to be used. However, when pulling a tree containing submodules, these will not be checked out by default; the *init* and *update* subcommands will maintain submodules checked out and at appropriate revision in your working tree. You can briefly inspect the up-to-date status of your submodules using the *status* subcommand and get a detailed overview of the difference between the index and checkouts using the *summary* subcommand.

COMMANDS

add

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".

This requires at least one argument: <repository>. The optional argument <path> is the relative location for the cloned submodule to exist in the superproject. If <path> is not given, the "humanish" part of the source repository is used ("repo" for "/path/to/repo.git" and "foo" for "host.xz:foo/.git"). The <path> is also used as the submodule’s logical name in its configuration entries unless `--name` is used to specify a logical name.

<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 origin 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). If the superproject doesn’t have an origin configured the superproject is its own authoritative upstream and the current working directory is used instead.

<path> is the relative location for the cloned submodule to exist in the superproject. If <path> does not exist, then the submodule is created by cloning from the named URL. If <path> does exist and is already a valid Git repository, then this is added to the changeset without cloning. This second form is provided to ease creating a new submodule from scratch, and presumes the user will later push the submodule to the given URL.

In either case, 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

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 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 `--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, git-status(1) and git-diff(1) will provide that information too (and can also report changes to a submodule’s work tree).

init

Initialize the submodules recorded in the index (which were added and committed elsewhere) by copying submodule names and urls from .gitmodules to .git/config. Optional <path> arguments limit which submodules will be initialized. It will also copy the value of `submodule.$name.update` into .git/config. The key used in .git/config is `submodule.$name.url`. This command does not alter existing information in .git/config. 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.

deinit

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 work tree anymore. If you really want to remove a submodule from the repository and commit that use git-rm(1) instead.

If `--force` is specified, the submodule’s work tree will be removed even if it contains local modifications.

update

Update the registered submodules to match what the superproject expects by cloning missing 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. Supported update procedures are:

checkout

the commit recorded in the superproject will be checked out in the submodule on a detached HEAD. This is done when `--checkout` option is given, or no option is given, and `submodule.<name>.update` is unset, or if it is set to *checkout*.

If `--force` is specified, the submodule will be checked out (using `git checkout --force` if appropriate), 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. This is done when `--rebase` option is given, or no option is given, and `submodule.<name>.update` is set to *rebase*.

merge

the commit recorded in the superproject will be merged into the current branch in the submodule. This is done when `--merge` option is given, or no option is given, and `submodule.<name>.update` is set to *merge*.

custom command

arbitrary shell command that takes a single argument (the sha1 of the commit recorded in the superproject) is executed. This is done when no option is given, and `submodule.<name>.update` has the form of *!command*.

When no option is given and `submodule.<name>.update` is set to *none*, the submodule is not updated.

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.

summary

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 git-diff(1) will provide that information too.

foreach

Evaluates an arbitrary shell command in each checked out submodule. The command has access to the variables $name, $path, $sha1 and $toplevel: $name is the name of the relevant submodule section in .gitmodules, $path is the name of the submodule directory relative to the superproject, $sha1 is the commit as recorded in the superproject, and $toplevel is the absolute path to the top-level of the superproject. 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, ``git submodule foreach 'echo $path `git rev-parse HEAD`'`` will show the path and currently checked out commit for each submodule.

sync

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.

OPTIONS

-q

--quiet

Only print error messages.

-b

--branch

Branch of repository to add as submodule. The name of the branch is recorded as `submodule.<name>.branch` in

`.gitmodules` for `update --remote`.

-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 work 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 `master`, 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 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 git-clone(1) command.

NOTE: Do **not** use this option unless you have read the note for git-clone(1)'s `--reference` and `--shared` options carefully.

--recursive
: This option is only valid for foreach, update and status 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 git-clone(1)

<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 gitmodules(5) for details.

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

git-svn(1) Manual Page

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-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-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*.

--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 an `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.

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.

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 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*:

```
        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>", which will be treated as if included in the authors file.

-q

--quiet
: Make *git svn* less verbose. Specify a second time to make it even less verbose.

-m

--merge

-s<strategy>

--strategy=<strategy>

-p

--preserve-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:` or `Signed-off-by:` line in the log message and use that as the author string.

--add-author-from

When committing to svn from Git (as part of *commit-diff*, *set-tree* or *dcommit* operations), if the existing log message doesn't already have a `From:` or `Signed-off-by:` line, 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.

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-branch(1) instead. filter-branch 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 git-push(1) documentation for details.

Do not use the --amend option of 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/*
```

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/*.

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/*
        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*.

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.

SEE ALSO

git-rebase(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-symbolic-ref(1) Manual Page

NAME

git-symbolic-ref - Read, modify and delete symbolic refs

SYNOPSIS

> *git symbolic-ref* [-m <reason>] <name> <ref>
> *git symbolic-ref* [-q] [--short] <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 contents 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`.

-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 git(1) suite

Last updated 2014-11-27 19:53:13 CET

git-tag(1) Manual Page

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>]
>   <tagname> [<commit> | <object>]
> *git tag* -d <tagname>…
> *git tag* [-n[<num>]] -l [--contains <commit>] [--points-at <object>]
>   [--column[=<options>] | --no-column] [<pattern>…]
>   [<pattern>…]
> *git tag* -v <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 just a tag reference for the SHA-1 object name of the commit object is created (i.e. a lightweight tag).

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.

-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. 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 <pattern>

--list <pattern>
: List tags with names that match the given pattern (or all if no pattern is given). 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.

--sort=<type>
: Sort in a specific order. Supported type is "refname" (lexicographic order), "version:refname" or "v:refname" (tag names are treated as versions). The "version:refname" sort order can also be affected by the "versionsort.prereleaseSuffix" configuration variable. Prepend "-" to reverse sort order. When this option is not given, the sort order defaults to the value configured for the *tag.sort* variable if it exists, or lexicographic order otherwise. See git-config(1).

--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).

--points-at <object>
: Only list tags of the given object.

-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.

--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.

<tagname>
: The name of the tag to create, delete, or describe. The new tag name must pass all checks defined by 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>
```

DISCUSSION

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.

On Automatic following

If you are following somebody else's tree, you are most likely using remote-tracking branches (`refs/heads/origin` in traditional layout, or `refs/remotes/origin/master` in the separate-remote layout). 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.

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, 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 2 hours ahead UTC) is `+0200`.

RFC 2822
: The standard email 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.

Note In addition, the date part is accepted in the following formats: `YYYY.MM.DD`, `MM/DD/YYYY` and `DD.MM.YYYY`.

SEE ALSO

git-check-ref-format(1). git-config(1).

GIT

Part of the git(1) suite

Last updated 2015-05-03 21:16:24 CEST

gitk(1) Manual Page

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 *sticked* form (see gitcli(7)) due to limitations in the command-line parser.

rev-list options and arguments

This manual page describes only the most frequently used options. See 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 diff 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 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 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 git-log(1) for a more detailed explanation.)

-L<start>,<end>:<file>

-L:<regex>:<file>
: Trace the evolution of the line range given by "<start>,<end>" (or the funcname regex <regex>) 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. You can specify this option more than once.

  Note: gitk (unlike git-log(1)) currently only understands this option if you specify it "glued together" with its argument. Do **not** put a space after `-L`.

  <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 ":<regex>" is given in place of <start> and <end>, it denotes the range from the first funcname line that matches <regex>, up to the next funcname line. “:<regex>” searches from the end of the previous `-L` range, if any, otherwise from the start of file. “^:<regex>” searches from the start of file.

<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 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.

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 and started off in a separate repository but was later merged into the main Git repository.

SEE ALSO

qgit(1)
: A repository browser written in C++ using Qt.

gitview(1)
: A repository browser written in Python using Gtk. It’s based on *bzrk(1)* and distributed in the contrib area of the Git repository.

tig(1)
: A minimal repository browser and Git tool output highlighter written in C using Ncurses.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:08 CET

git-unpack-file(1) Manual Page

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 git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-unpack-objects(1) Manual Page

NAME

git-unpack-objects - Unpack objects from a packed archive

SYNOPSIS

> *git unpack-objects* [-n] [-q] [-r] [--strict] < <pack-file>

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 pack-file. Therefore, nothing will be unpacked if you use this command on a pack-file that exists within the target repository.

See 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.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-update-index(1) Manual Page

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]
          [--ignore-submodules]
          [--really-refresh] [--unresolve] [--again | -g]
          [--info-only] [--index-info]
          [-z] [--stdin] [--index-version <n>]
          [--verbose]
          [--] [<file>...]
```

DESCRIPTION

Modifies the index or directory cache. Each file mentioned is updated into the index and any *unmerged* or *needs updating* state is cleared.

See also 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 file.

--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 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 name 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.

-g

--again
: Runs *git update-index* itself on the paths whose index entries are different from those from 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 list of paths from the command line, read 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 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`.

  Version 4 performs a simple pathname compression that reduces index size by 30%-50% on large repositories, which results in faster load time. Version 4 is relatively young (first released in in 1.8.0 in October 2012). Other Git implementations such as JGit and libgit2 may not support it yet.

-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 enabled, the index is split into two files, $GIT_DIR/index and $GIT_DIR/sharedindex.<SHA-1>. Changes are accumulated in $GIT_DIR/index while the shared index file contains all index entries stays unchanged. 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. This mode is designed for very large indexes that take a significant amount of time to read or write.

--
: 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 with mode and sha1 at path, say:

```
$ git update-index --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 sha1 TAB path

   The first format is what "git-apply --index-info" reports, and used to reconstruct a partial tree that is used for phony merge base tree when falling back on 3-way merge.

2. mode SP type SP sha1 TAB path

   The second format is to stuff *git ls-tree* output into the index file.

3. 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.

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 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").

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              <1>
$ git update-index --no-assume-unchanged foo.c   <2>
$ git diff --name-only                           <3>
$ edit foo.c
$ git diff --name-only                           <4>
M foo.c
$ git update-index foo.c                         <5>
$ git diff --name-only                           <6>
$ edit foo.c
$ git diff --name-only                           <7>
$ git update-index --no-assume-unchanged foo.c   <8>
$ git diff --name-only                           <9>
M foo.c
```

1. forces lstat(2) to set "assume unchanged" bits for paths that match index.
2. mark the path to be edited.
3. this does lstat(2) and finds index matches the path.
4. this does lstat(2) and finds index does **not** match the path.
5. registering the new version to index sets "assume unchanged" bit.
6. and it is assumed unchanged.
7. even after you edit it.
8. you can tell about the change after the fact.
9. 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: When reading an entry, if it is marked as skip-worktree, then Git pretends its working directory version is up to date and read the index version instead.

To elaborate, "reading" means checking for file existence, reading file attributes or file content. The working directory version may be present or absent. If present, its content may match against the index version or not. Writing is not affected by this bit, content safety is still first priority. Note that Git *can* update working directory file, that is marked skip-worktree, if it is safe to do so (i.e. working directory version matches index version)

Although this bit looks similar to assume-unchanged bit, its goal is different from assume-unchanged bit's. Skip-worktree also takes precedence over assume-unchanged bit when both are set.

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 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 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 git-config(1)).

SEE ALSO

git-config(1), git-add(1), git-ls-files(1)

GIT

Part of the git(1) suite

Last updated 2014-12-23 17:41:32 CET

git-update-ref(1) Manual Page

NAME

git-update-ref - Update the object name stored in a ref safely

SYNOPSIS

> *git update-ref* [-m <reason>] (-d <ref> [<oldvalue>] | [--no-deref] <ref> <newvalue> [<oldvalue>] | --stdin [-z])

DESCRIPTION

Given two arguments, stores the <newvalue> in the <ref>, possibly dereferencing the symbolic refs. E.g. `git update-ref HEAD <newvalue>` updates the current branch head to the new object.

Given three arguments, stores the <newvalue> in the <ref>, possibly dereferencing the symbolic refs, after verifying that the current value of the <ref> matches <oldvalue>. E.g. `git update-ref refs/heads/master <newvalue> <oldvalue>` updates the master branch head to <newvalue> only if its current value is <oldvalue>. You can specify 40 "0" or an empty string as <oldvalue> 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 <oldvalue>.

With `--stdin`, update-ref reads instructions from standard input and performs all modifications together. Specify commands of the form:

```
update SP <ref> SP <newvalue> [SP <oldvalue>] LF
create SP <ref> SP <newvalue> LF
delete SP <ref> [SP <oldvalue>] LF
verify SP <ref> [SP <oldvalue>] LF
option SP <opt> LF
```

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 <newvalue> NUL [<oldvalue>] NUL
create SP <ref> NUL <newvalue> NUL
delete SP <ref> NUL [<oldvalue>] NUL
verify SP <ref> NUL [<oldvalue>] NUL
option SP <opt> 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 <newvalue> after verifying <oldvalue>, if given. Specify a zero <newvalue> to ensure the ref does not exist after the update and/or a zero <oldvalue> to make sure the ref does not exist before the update.

create
: Create <ref> with <newvalue> after verifying it does not exist. The given <newvalue> may not be zero.

delete
: Delete <ref> after verifying it exists with <oldvalue>, if given. If given, <oldvalue> may not be zero.

verify
: Verify <ref> against <oldvalue> but do not change it. If <oldvalue> zero or missing, the ref must not exist.

option
: Modify behavior of the next command naming a <ref>. The only valid option is `no-deref` to avoid dereferencing a symbolic ref.

If all <ref>s can be locked with matching <oldvalue>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 the symbolic ref 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:

1. 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 <newvalue> and "committer" is the committer's name, email address and date in the standard Git committer ident format.

Optionally with -m:

1. 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 git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-update-server-info(1) Manual Page

NAME

git-update-server-info - Update auxiliary info file to help dumb servers

SYNOPSIS

> *git update-server-info* [--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 gitrepository-layout(5) for description of what they are for:

- objects/info/packs
- info/refs

GIT

Part of the git(1) suite

Last updated 2014-01-25 09:03:55 CET

git-upload-archive(1) Manual Page

NAME

git-upload-archive - Send archive back to git-archive

SYNOPSIS

> *git upload-archive* <directory>

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

<directory>
: The repository to get a tar archive from.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

git-upload-pack(1) Manual Page

NAME

git-upload-pack - Send objects packed back to git-fetch-pack

SYNOPSIS

> *git-upload-pack* [--strict] [--timeout=<n>] <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

--strict
: Do not try <directory>/.git/ if <directory> is no Git directory.

--timeout=<n>
: Interrupt transfer after <n> seconds of inactivity.

<directory>
: The repository to sync from.

SEE ALSO

gitnamespaces(7)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-var(1) Manual Page

NAME

git-var - Show a Git logical variable

SYNOPSIS

> *git var* (-l | <variable>)

DESCRIPTION

Prints a Git logical variable.

OPTIONS

-l
: Cause the logical variables to be listed. 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`.)

EXAMPLE

```
$ 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_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*).

SEE ALSO

git-commit-tree(1) git-tag(1) git-config(1)

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:54:14 CET

git-verify-commit(1) Manual Page

NAME

git-verify-commit - Check the GPG signature of commits

SYNOPSIS

> *git verify-commit* <commit>…

DESCRIPTION

Validates the gpg signature created by *git commit -S*.

OPTIONS

-v

--verbose

Print the contents of the commit object before validating it.

<commit>…

SHA-1 identifiers of Git commit objects.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-verify-pack(1) Manual Page

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 idx file and the corresponding pack file.

OPTIONS

<pack>.idx …

The idx files to verify.

-v

--verbose

After verifying the pack, show 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`, 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-pack-file 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 git(1) suite

Last updated 2014-11-27 19:55:05 CET

git-verify-tag(1) Manual Page

NAME

git-verify-tag - Check the GPG signature of tags

SYNOPSIS

> *git verify-tag* <tag>…

DESCRIPTION

Validates the gpg signature created by *git tag*.

OPTIONS

-v

--verbose
: Print the contents of the tag object before validating it.

<tag>…
: SHA-1 identifiers of Git tag objects.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:55:05 CET

git-web--browse(1) Manual Page

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

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.

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.

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
```

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 git-config(1) for more information about this.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:07 CET

git-whatchanged(1) Manual Page

NAME

git-whatchanged - Show logs with difference each commit introduces

SYNOPSIS

> *git whatchanged* <option>…

DESCRIPTION

Shows commit logs and diff output each commit introduces.

New users are encouraged to use git-log(1) instead. The `whatchanged` command is essentially the same as git-log(1) but defaults to show the raw format diff output and to skip merges.

The command is kept primarily for historical reasons; fingers of many people who learned Git long before `git log` was invented by reading 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 git(1) suite

Last updated 2014-11-27 19:56:10 CET

git-write-tree(1) Manual Page

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 git(1) suite

Last updated 2014-01-25 09:03:55 CET

gitattributes(5) Manual Page

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. 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 how the pattern matches paths are the same as in `.gitignore` files; see gitignore(5). Unlike `.gitignore`, negative patterns are forbidden.

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 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 an 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 `!`.

EFFECTS

Certain operations by Git can be influenced by assigning particular attributes to a path. Currently, the following operations are attributes-aware.

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 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*.

`text`

This attribute enables and controls end-of-line normalization. When a text file is normalized, its line endings are converted to LF in the repository. To control what line ending style is used in the working directory, use the `eol` attribute for a single file and the `core.eol` configuration variable for all text files.

Set
: Setting the `text` attribute on a path enables end-of-line normalization and marks the path as a text file. End-of-line conversion takes place without guessing the content type.

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", the path is marked for automatic end-of-line normalization. If Git decides that the content is text, its line endings are normalized to LF on checkin.

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.

`eol`

This attribute sets a specific line-ending style to be used in the working directory. It enables end-of-line normalization without any content checks, effectively setting the `text` attribute.

Set to string value "crlf"
: This setting forces Git to normalize line endings for this file on checkin and convert them to CRLF when the file is checked out.

Set to string value "lf"
: This setting forces Git to normalize line endings to LF on checkin and prevents conversion to CRLF when the file is checked out.

Backwards compatibility with `crlf` attribute

For backwards compatibility, the `crlf` attribute is interpreted as follows:

```
crlf            text
-crlf           -text
crlf=input      eol=lf
```

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.

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.

```
*.txt           text
*.vcproj        eol=crlf
*.sh            eol=lf
*.jpg           -text
```

Other source code management systems normalize all text files in their repositories, and there are two ways to enable similar automatic normalization in Git.

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 changing any attributes.

```
[core]
        autocrlf = true
```

This does not force normalization of all 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 interoperate with a source code management system that enforces end-of-line normalization, or you simply want all text files in your repository to be normalized, you should instead set the `text` attribute to "auto" for *all* files.

```
*       text=auto
```

This ensures that all files that Git considers to be text will have normalized (LF) line endings in the repository. The `core.eol` configuration variable controls which line endings Git will use for normalized files in your working directory; the default is to use the native line ending for your platform, or CRLF if `core.autocrlf` is set.

Note When `text=auto` normalization is enabled in an existing repository, any text files containing CRLFs should be normalized. If they are not they will be normalized the next time someone tries to change them, causing unfortunate misattribution. From a clean working directory:

```
$ echo "* text=auto" >>.gitattributes
$ rm .git/index     # Remove the index to force Git to
$ git reset         # re-scan the working directory
$ git status        # Show files that will be normalized
$ git add -u
$ git add .gitattributes
$ 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.

`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.

`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.

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`.

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
```

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`.

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.

Generating diff text

`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.

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 git(1) for details.

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.
- `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.
- `fortran` suitable for source code in the Fortran language.
- `html` suitable for HTML/XHTML documents.
- `java` suitable for source code in the Java language.
- `matlab` suitable for source code in the MATLAB language.
- `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.
- `tex` suitable for source code for LaTeX documents.

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.

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).

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`.

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
```

Performing a three-way merge

`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".

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.

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
        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 merge 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.

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.

`conflict-marker-size`

This attribute controls the length of conflict markers left in the work tree file during a conflicted merge. Only setting to the value to a positive integer has any 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
```

Checking whitespace errors

`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 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 separate list of common whitespace problems to notice in the same format as the `core.whitespace` configuration variable.

Creating an archive

`export-ignore`

Files and directories with the attribute `export-ignore` won't be added to archive files.

`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 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 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.

Packing objects

`delta`

Delta compression will not be attempted for blobs for paths with the attribute `delta` set to false.

Viewing files in GUI tools

`encoding`

The value of this attribute specifies the character encoding that should be used by GUI tools (e.g. gitk(1) and git-gui(1)) to display the contents of the relevant file. Note that due to performance considerations 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 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
```

EXAMPLE

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

git-check-attr(1).

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

Fighting regressions with git bisect

Christian Couder
<chriscool@tuxfamily.org>
2009/11/08

Abstract

"git bisect" enables software users and developers to easily find the commit that introduced a regression. We show why it is important to have good tools to fight regressions. We describe how "git bisect" works from the outside and the algorithms it uses inside. Then we explain how to take advantage of "git bisect" to improve current practices. And we discuss how "git bisect" could improve in the future.

Introduction to "git bisect"

Git is a Distributed Version Control system (DVCS) created by Linus Torvalds and maintained by Junio Hamano.

In Git like in many other Version Control Systems (VCS), the different states of the data that is managed by the system are called commits. And, as VCS are mostly used to manage software source code, sometimes "interesting" changes of behavior in the software are introduced in some commits.

In fact people are specially interested in commits that introduce a "bad" behavior, called a bug or a regression. They are interested in these commits because a commit (hopefully) contains a very small set of source code changes. And it’s much easier to understand and properly fix a problem when you only need to check a very small set of changes, than when you don’t know where look in the first place.

So to help people find commits that introduce a "bad" behavior, the "git bisect" set of commands was invented. And it follows of course that in "git bisect" parlance, commits where the "interesting behavior" is present are called "bad" commits, while other commits are called "good" commits. And a commit that introduce the behavior we are interested in is called a "first bad commit". Note that there could be more than one "first bad commit" in the commit space we are searching.

So "git bisect" is designed to help find a "first bad commit". And to be as efficient as possible, it tries to perform a binary search.

Fighting regressions overview

Regressions: a big problem

Regressions are a big problem in the software industry. But it’s difficult to put some real numbers behind that claim.

There are some numbers about bugs in general, like a NIST study in 2002 [1] that said:

> Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product, according to a newly released study commissioned by the Department of Commerce’s National Institute of Standards and Technology (NIST). At the national level, over half of the costs are borne by software users and the remainder by software developers/vendors. The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects. These are the savings associated with finding an increased percentage (but not 100 percent) of errors closer to the development stages in which they are introduced. Currently, over half of all errors are not found until "downstream" in the development process or during post-sale software use.

And then:

> Software developers already spend approximately 80 percent of development costs on identifying and correcting defects, and yet few products of any type other than software are shipped with such high levels of errors.

Eventually the conclusion started with:

> The path to higher software quality is significantly improved software testing.

There are other estimates saying that 80% of the cost related to software is about maintenance [2].

Though, according to Wikipedia [3]:

> A common perception of maintenance is that it is merely fixing bugs. However, studies and surveys over the years have indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions (Pigosky 1997). This perception is perpetuated by users submitting problem reports that in reality are functionality enhancements to the system.

But we can guess that improving on existing software is very costly because you have to watch out for regressions. At least this would make the above studies consistent among themselves.

Of course some kind of software is developed, then used during some time without being improved on much, and then finally thrown away. In this case, of course, regressions may not be a big problem. But on the other hand, there is a lot of big software that is continually developed and maintained during years or even tens of years by a lot of people. And as there are often many people who depend (sometimes critically) on such software, regressions are a really big problem.

One such software is the Linux kernel. And if we look at the Linux kernel, we can see that a lot of time and effort is spent to fight regressions. The release cycle start with a 2 weeks long merge window. Then the first release candidate (rc) version is tagged. And after that about 7 or 8 more rc versions will appear with around one week between each of them, before the final release.

The time between the first rc release and the final release is supposed to be used to test rc versions and fight bugs and especially regressions. And this time is more than 80% of the release cycle time. But this is not the end of the fight yet, as of course it continues after the release.

And then this is what Ingo Molnar (a well known Linux kernel developer) says about his use of git bisect:

> I most actively use it during the merge window (when a lot of trees get merged upstream and when the influx of bugs is the highest) - and yes, there have been cases that i used it multiple times a day. My average is roughly once a day.

So regressions are fought all the time by developers, and indeed it is well known that bugs should be fixed as soon as possible, so as soon as they are found. That's why it is interesting to have good tools for this purpose.

Other tools to fight regressions

So what are the tools used to fight regressions? They are nearly the same as those used to fight regular bugs. The only specific tools are test suites and tools similar as "git bisect".

Test suites are very nice. But when they are used alone, they are supposed to be used so that all the tests are checked after each commit. This means that they are not very efficient, because many tests are run for no interesting result, and they suffer from combinational explosion.

In fact the problem is that big software often has many different configuration options and that each test case should pass for each configuration after each commit. So if you have for each release: N configurations, M commits and T test cases, you should perform:

```
N * M * T tests
```

where N, M and T are all growing with the size your software.

So very soon it will not be possible to completely test everything.

And if some bugs slip through your test suite, then you can add a test to your test suite. But if you want to use your new improved test suite to find where the bug slipped in, then you will either have to emulate a bisection process or you will perhaps bluntly test each commit backward starting from the "bad" commit you have which may be very wasteful.

"git bisect" overview

Starting a bisection

The first "git bisect" subcommand to use is "git bisect start" to start the search. Then bounds must be set to limit the commit space. This is done usually by giving one "bad" and at least one "good" commit. They can be passed in the initial call to "git bisect start" like this:

```
$ git bisect start [BAD [GOOD...]]
```

or they can be set using:

```
$ git bisect bad [COMMIT]
```

and:

```
$ git bisect good [COMMIT...]
```

where BAD, GOOD and COMMIT are all names that can be resolved to a commit.

Then "git bisect" will checkout a commit of its choosing and ask the user to test it, like this:

```
$ git bisect start v2.6.27 v2.6.25
Bisecting: 10928 revisions left to test after this (roughly 14 steps)
[2ec65f8b89ea003c27ff7723525a2ee335a2b393] x86: clean up using max_low_pfn on 32-bit
```

Note that the example that we will use is really a toy example, we will be looking for the first commit that has a version like "2.6.26-something", that is the commit that has a "SUBLEVEL = 26" line in the top level Makefile. This is a toy example because there are better ways to find this commit with Git than using "git bisect" (for example "git blame" or "git log -S<string>").

Driving a bisection manually

At this point there are basically 2 ways to drive the search. It can be driven manually by the user or it can be driven automatically by a script or a command.

If the user is driving it, then at each step of the search, the user will have to test the current commit and say if it is "good" or "bad" using the "git bisect good" or "git bisect bad" commands respectively that have been described above. For example:

```
$ git bisect bad
Bisecting: 5480 revisions left to test after this (roughly 13 steps)
[66c0b394f08fd89236515c1c84485ea712a157be] KVM: kill file->f_count abuse in kvm
```

And after a few more steps like that, "git bisect" will eventually find a first bad commit:

```
$ git bisect bad
2ddcca36c8bcfa251724fe342c8327451988be0d is the first bad commit
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Sat May 3 11:59:44 2008 -0700

    Linux 2.6.26-rc1

:100644 100644 5cf82581... 4492984e... M      Makefile
```

At this point we can see what the commit does, check it out (if it’s not already checked out) or tinker with it, for example:

```
$ git show HEAD
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Sat May 3 11:59:44 2008 -0700

    Linux 2.6.26-rc1

diff --git a/Makefile b/Makefile
index 5cf8258..4492984 100644
--- a/Makefile
+++ b/Makefile
@@ -1,7 +1,7 @@
 VERSION = 2
 PATCHLEVEL = 6
-SUBLEVEL = 25
-EXTRAVERSION =
+SUBLEVEL = 26
+EXTRAVERSION = -rc1
 NAME = Funky Weasel is Jiggy wit it

 # *DOCUMENTATION*
```

And when we are finished we can use "git bisect reset" to go back to the branch we were in before we started bisecting:

```
$ git bisect reset
Checking out files: 100% (21549/21549), done.
Previous HEAD position was 2ddcca3... Linux 2.6.26-rc1
Switched to branch 'master'
```

Driving a bisection automatically

The other way to drive the bisection process is to tell "git bisect" to launch a script or command at each bisection step to know if the current commit is "good" or "bad". To do that, we use the "git bisect run" command. For example:

```
$ git bisect start v2.6.27 v2.6.25
Bisecting: 10928 revisions left to test after this (roughly 14 steps)
[2ec65f8b89ea003c27ff7723525a2ee335a2b393] x86: clean up using max_low_pfn on 32-bit
$
$ git bisect run grep '^SUBLEVEL = 25' Makefile
running grep ^SUBLEVEL = 25 Makefile
Bisecting: 5480 revisions left to test after this (roughly 13 steps)
[66c0b394f08fd89236515c1c84485ea712a157be] KVM: kill file->f_count abuse in kvm
running grep ^SUBLEVEL = 25 Makefile
SUBLEVEL = 25
Bisecting: 2740 revisions left to test after this (roughly 12 steps)
[671294719628f1671faefd4882764886f8ad08cb] V4L/DVB(7879): Adding cx18 Support for mxl5005s
...
...
running grep ^SUBLEVEL = 25 Makefile
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[2ddcca36c8bcfa251724fe342c8327451988be0d] Linux 2.6.26-rc1
running grep ^SUBLEVEL = 25 Makefile
2ddcca36c8bcfa251724fe342c8327451988be0d is the first bad commit
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Sat May 3 11:59:44 2008 -0700

    Linux 2.6.26-rc1

:100644 100644 5cf82581... 4492984e... M      Makefile
bisect run success
```

In this example, we passed "grep *^SUBLEVEL = 25* Makefile" as parameter to "git bisect run". This means that at each step, the grep command we passed will be launched. And if it exits with code 0 (that means success) then git bisect will mark the current state as "good". If it exits with code 1 (or any code between 1 and 127 included, except the special code 125), then the current state will be marked as "bad".

Exit code between 128 and 255 are special to "git bisect run". They make it stop immediately the bisection process. This is useful for example if the command passed takes too long to complete, because you can kill it with a signal and it will stop the bisection process.

It can also be useful in scripts passed to "git bisect run" to "exit 255" if some very abnormal situation is detected.

Avoiding untestable commits

Sometimes it happens that the current state cannot be tested, for example if it does not compile because there was a bug preventing it at that time. This is what the special exit code 125 is for. It tells "git bisect run" that the current commit should be marked as untestable and that another one should be chosen and checked out.

If the bisection process is driven manually, you can use "git bisect skip" to do the same thing. (In fact the special exit code 125 makes "git bisect run" use "git bisect skip" in the background.)

Or if you want more control, you can inspect the current state using for example "git bisect visualize". It will launch gitk (or "git log" if the DISPLAY environment variable is not set) to help you find a better bisection point.

Either way, if you have a string of untestable commits, it might happen that the regression you are looking for has been introduced by one of these untestable commits. In this case it’s not possible to tell for sure which commit introduced the regression.

So if you used "git bisect skip" (or the run script exited with special code 125) you could get a result like this:

```
There are only 'skip'ped commits left to test.
The first bad commit could be any of:
15722f2fa328eaba97022898a305ffc8172db6b1
78e86cf3e850bd755bb71831f42e200626fbd1e0
e15b73ad3db9b48d7d1ade32f8cd23a751fe0ace
070eab2303024706f2924822bfec8b9847e4ac1b
We cannot bisect more!
```

Saving a log and replaying it

If you want to show other people your bisection process, you can get a log using for example:

```
$ git bisect log > bisect_log.txt
```

And it is possible to replay it using:

```
$ git bisect replay bisect_log.txt
```

"git bisect" details

Bisection algorithm

As the Git commits form a directed acyclic graph (DAG), finding the best bisection commit to test at each step is not so simple. Anyway Linus found and implemented a "truly stupid" algorithm, later improved by Junio Hamano, that works quite well.

So the algorithm used by "git bisect" to find the best bisection commit when there are no skipped commits is the following:

1) keep only the commits that:

a) are ancestor of the "bad" commit (including the "bad" commit itself), b) are not ancestor of a "good" commit (excluding the "good" commits).

This means that we get rid of the uninteresting commits in the DAG.

For example if we start with a graph like this:

```
G-Y-G-W-W-W-X-X-X-X
           \ /
            W-W-B
           /
Y---G-W---W
 \ /   \
Y-Y     X-X-X-X

-> time goes this way ->
```

where B is the "bad" commit, "G" are "good" commits and W, X, and Y are other commits, we will get the following graph after this first step:

```
W-W-W
     \
      W-W-B
     /
W---W
```

So only the W and B commits will be kept. Because commits X and Y will have been removed by rules a) and b) respectively, and because commits G are removed by rule b) too.

Note for Git users, that it is equivalent as keeping only the commit given by:

```
git rev-list BAD --not GOOD1 GOOD2...
```

Also note that we don’t require the commits that are kept to be descendants of a "good" commit. So in the following example, commits W and Z will be kept:

```
G-W-W-W-B
   /
Z-Z
```

2) starting from the "good" ends of the graph, associate to each commit the number of ancestors it has plus one

For example with the following graph where H is the "bad" commit and A and D are some parents of some "good" commits:

```
A-B-C
     \
      F-G-H
     /
D---E
```

this will give:

```
1 2 3
A-B-C
     \6 7 8
      F-G-H
1   2/
D---E
```

3) associate to each commit: min(X, N - X)

where X is the value associated to the commit in step 2) and N is the total number of commits in the graph.

In the above example we have N = 8, so this will give:

```
1 2 3
A-B-C
     \2 1 0
      F-G-H
1   2/
D---E
```

4) the best bisection point is the commit with the highest associated number

So in the above example the best bisection point is commit C.

5) note that some shortcuts are implemented to speed up the algorithm

As we know N from the beginning, we know that min(X, N - X) can’t be greater than N/2. So during steps 2) and 3), if we would associate N/2 to a commit, then we know this is the best bisection point. So in this case we can just stop processing any other commit and return the current commit.

Bisection algorithm debugging

For any commit graph, you can see the number associated with each commit using "git rev-list --bisect-all".

For example, for the above graph, a command like:

```
$ git rev-list --bisect-all BAD --not GOOD1 GOOD2
```

would output something like:

```
e15b73ad3db9b48d7d1ade32f8cd23a751fe0ace (dist=3)
15722f2fa328eaba97022898a305ffc8172db6b1 (dist=2)
78e86cf3e850bd755bb71831f42e200626fbd1e0 (dist=2)
a1939d9a142de972094af4dde9a544e577ddef0e (dist=2)
070eab2303024706f2924822bfec8b9847e4ac1b (dist=1)
a3864d4f32a3bf5ed177ddef598490a08760b70d (dist=1)
a41baa717dd74f1180abf55e9341bc7a0bb9d556 (dist=1)
9e622a6dad403b71c40979743bb9d5be17b16bd6 (dist=0)
```

Bisection algorithm discussed

First let’s define "best bisection point". We will say that a commit X is a best bisection point or a best bisection commit if knowing its state ("good" or "bad") gives as much information as possible whether the state of the commit happens to be "good" or "bad".

This means that the best bisection commits are the commits where the following function is maximum:

```
f(X) = min(information_if_good(X), information_if_bad(X))
```

where information_if_good(X) is the information we get if X is good and information_if_bad(X) is the information we get if X is bad.

Now we will suppose that there is only one "first bad commit". This means that all its descendants are "bad" and all the other commits are "good". And we will suppose that all commits have an equal probability of being good or bad, or of being the first bad commit, so knowing the state of c commits gives always the same amount of information wherever these c commits are on the graph and whatever c is. (So we suppose that these commits being for example on a branch or near a good or a bad commit does not give more or less information).

Let’s also suppose that we have a cleaned up graph like one after step 1) in the bisection algorithm above. This means that we can measure the information we get in terms of number of commit we can remove from the graph..

And let’s take a commit X in the graph.

If X is found to be "good", then we know that its ancestors are all "good", so we want to say that:

```
information_if_good(X) = number_of_ancestors(X)  (TRUE)
```

And this is true because at step 1) b) we remove the ancestors of the "good" commits.

If X is found to be "bad", then we know that its descendants are all "bad", so we want to say that:

```
information_if_bad(X) = number_of_descendants(X)  (WRONG)
```

But this is wrong because at step 1) a) we keep only the ancestors of the bad commit. So we get more information when a commit is marked as "bad", because we also know that the ancestors of the previous "bad" commit that are not ancestors of the new "bad" commit are not the first bad commit. We don’t know if they are good or bad, but we know that they are not the first bad commit because they are not ancestor of the new "bad" commit.

So when a commit is marked as "bad" we know we can remove all the commits in the graph except those that are ancestors of the new "bad" commit. This means that:

```
information_if_bad(X) = N - number_of_ancestors(X)  (TRUE)
```

where N is the number of commits in the (cleaned up) graph.

So in the end this means that to find the best bisection commits we should maximize the function:

```
f(X) = min(number_of_ancestors(X), N - number_of_ancestors(X))
```

And this is nice because at step 2) we compute number_of_ancestors(X) and so at step 3) we compute f(X).

Let’s take the following graph as an example:

```
            G-H-I-J
           /       \
A-B-C-D-E-F         O
           \       /
            K-L-M-N
```

If we compute the following non optimal function on it:

```
g(X) = min(number_of_ancestors(X), number_of_descendants(X))
```

we get:

```
            4 3 2 1
            G-H-I-J
1 2 3 4 5 6/       \0
A-B-C-D-E-F         O
           \       /
            K-L-M-N
            4 3 2 1
```

but with the algorithm used by git bisect we get:

```
            7 7 6 5
            G-H-I-J
1 2 3 4 5 6/       \0
A-B-C-D-E-F         O
           \       /
            K-L-M-N
            7 7 6 5
```

So we chose G, H, K or L as the best bisection point, which is better than F. Because if for example L is bad, then we will know not only that L, M and N are bad but also that G, H, I and J are not the first bad commit (since we suppose that there is only one first bad commit and it must be an ancestor of L).

So the current algorithm seems to be the best possible given what we initially supposed.

Skip algorithm

When some commits have been skipped (using "git bisect skip"), then the bisection algorithm is the same for step 1) to 3). But then we use roughly the following steps:

6) sort the commit by decreasing associated value

7) if the first commit has not been skipped, we can return it and stop here

8) otherwise filter out all the skipped commits in the sorted list

9) use a pseudo random number generator (PRNG) to generate a random number between 0 and 1

10) multiply this random number with its square root to bias it toward 0

11) multiply the result by the number of commits in the filtered list to get an index into this list

12) return the commit at the computed index

Skip algorithm discussed

After step 7) (in the skip algorithm), we could check if the second commit has been skipped and return it if it is not the case. And in fact that was the algorithm we used from when "git bisect skip" was developed in Git version 1.5.4 (released on February 1st 2008) until Git version 1.6.4 (released July 29th 2009).

But Ingo Molnar and H. Peter Anvin (another well known linux kernel developer) both complained that sometimes the best bisection points all happened to be in an area where all the commits are untestable. And in this case the user was asked to test many untestable commits, which could be very inefficient.

Indeed untestable commits are often untestable because a breakage was introduced at one time, and that breakage was fixed only after many other commits were introduced.

This breakage is of course most of the time unrelated to the breakage we are trying to locate in the commit graph. But it prevents us to know if the interesting "bad behavior" is present or not.

So it is a fact that commits near an untestable commit have a high probability of being untestable themselves. And the best bisection commits are often found together too (due to the bisection algorithm).

This is why it is a bad idea to just chose the next best unskipped bisection commit when the first one has been skipped.

We found that most commits on the graph may give quite a lot of information when they are tested. And the commits that will not on average give a lot of information are the one near the good and bad commits.

So using a PRNG with a bias to favor commits away from the good and bad commits looked like a good choice.

One obvious improvement to this algorithm would be to look for a commit that has an associated value near the one of the best bisection commit, and that is on another branch, before using the PRNG. Because if such a commit exists, then it is not very likely to be untestable too, so it will probably give more information than a nearly randomly chosen one.

Checking merge bases

There is another tweak in the bisection algorithm that has not been described in the "bisection algorithm" above.

We supposed in the previous examples that the "good" commits were ancestors of the "bad" commit. But this is not a requirement of "git bisect".

Of course the "bad" commit cannot be an ancestor of a "good" commit, because the ancestors of the good commits are supposed to be "good". And all the "good" commits must be related to the bad commit. They cannot be on a branch that has no link with the branch of the "bad" commit. But it is possible for a good commit to be related to a bad commit and yet not be neither one of its ancestor nor one of its descendants.

For example, there can be a "main" branch, and a "dev" branch that was forked of the main branch at a commit named "D" like this:

```
A-B-C-D-E-F-G  <--main
       \
        H-I-J  <--dev
```

The commit "D" is called a "merge base" for branch "main" and "dev" because it’s the best common ancestor for these branches for a merge.

Now let’s suppose that commit J is bad and commit G is good and that we apply the bisection algorithm like it has been previously described.

As described in step 1) b) of the bisection algorithm, we remove all the ancestors of the good commits because they are supposed to be good too.

So we would be left with only:

```
H-I-J
```

But what happens if the first bad commit is "B" and if it has been fixed in the "main" branch by commit "F"?

The result of such a bisection would be that we would find that H is the first bad commit, when in fact it’s B. So that would be wrong!

And yes it can happen in practice that people working on one branch are not aware that people working on another branch fixed a bug! It could also happen that F fixed more than one bug or that it is a revert of some big development effort that was not ready to be released.

In fact development teams often maintain both a development branch and a maintenance branch, and it would be quite easy for them if "git bisect" just worked when they want to bisect a regression on the development branch that is not on the maintenance branch. They should be able to start bisecting using:

```
$ git bisect start dev main
```

To enable that additional nice feature, when a bisection is started and when some good commits are not ancestors of the bad commit, we first compute the merge bases between the bad and the good commits and we chose these merge bases as the first commits that will be checked out and tested.

If it happens that one merge base is bad, then the bisection process is stopped with a message like:

```
The merge base BBBBBB is bad.
This means the bug has been fixed between BBBBBB and [GGGGGG,...].
```

where BBBBBB is the sha1 hash of the bad merge base and [GGGGGG,...] is a comma separated list of the sha1 of the good commits.

If some of the merge bases are skipped, then the bisection process continues, but the following message is printed for each skipped merge base:

```
Warning: the merge base between BBBBBB and [GGGGGG,...] must be skipped.
So we cannot be sure the first bad commit is between MMMMMM and BBBBBB.
We continue anyway.
```

where BBBBBB is the sha1 hash of the bad commit, MMMMMM is the sha1 hash of the merge base that is skipped and [GGGGGG,...] is a comma separated list of the sha1 of the good commits.

So if there is no bad merge base, the bisection process continues as usual after this step.

Best bisecting practices

Using test suites and git bisect together

If you both have a test suite and use git bisect, then it becomes less important to check that all tests pass after each commit. Though of course it is probably a good idea to have some checks to avoid breaking too many things because it could make bisecting other bugs more difficult.

You can focus your efforts to check at a few points (for example rc and beta releases) that all the T test cases pass for all the N configurations. And when some tests don't pass you can use "git bisect" (or better "git bisect run"). So you should perform roughly:

```
c * N * T + b * M * log2(M) tests
```

where c is the number of rounds of test (so a small constant) and b is the ratio of bug per commit (hopefully a small constant too).

So of course it's much better as it's O(N * T) vs O(N * T * M) if you would test everything after each commit.

This means that test suites are good to prevent some bugs from being committed and they are also quite good to tell you that you have some bugs. But they are not so good to tell you where some bugs have been introduced. To tell you that efficiently, git bisect is needed.

The other nice thing with test suites, is that when you have one, you already know how to test for bad behavior. So you can use this knowledge to create a new test case for "git bisect" when it appears that there is a regression. So it will be easier to bisect the bug and fix it. And then you can add the test case you just created to your test suite.

So if you know how to create test cases and how to bisect, you will be subject to a virtuous circle:

more tests ⇒ easier to create tests ⇒ easier to bisect ⇒ more tests

So test suites and "git bisect" are complementary tools that are very powerful and efficient when used together.

Bisecting build failures

You can very easily automatically bisect broken builds using something like:

```
$ git bisect start BAD GOOD
$ git bisect run make
```

Passing sh -c "some commands" to "git bisect run"

For example:

```
$ git bisect run sh -c "make || exit 125; ./my_app | grep 'good output'"
```

On the other hand if you do this often, then it can be worth having scripts to avoid too much typing.

Finding performance regressions

Here is an example script that comes slightly modified from a real world script used by Junio Hamano [4].

This script can be passed to "git bisect run" to find the commit that introduced a performance regression:

```
#!/bin/sh

# Build errors are not what I am interested in.
make my_app || exit 255

# We are checking if it stops in a reasonable amount of time, so
# let it run in the background...

./my_app >log 2>&1 &

# ... and grab its process ID.
pid=$!

# ... and then wait for sufficiently long.
sleep $NORMAL_TIME

# ... and then see if the process is still there.
if kill -0 $pid
then
        # It is still running -- that is bad.
        kill $pid; sleep 1; kill $pid;
        exit 1
else
        # It has already finished (the $pid process was no more),
        # and we are happy.
        exit 0
fi
```

Following general best practices

It is obviously a good idea not to have commits with changes that knowingly break things, even if some other commits later fix the breakage.

It is also a good idea when using any VCS to have only one small logical change in each commit.

The smaller the changes in your commit, the most effective "git bisect" will be. And you will probably need "git bisect" less in the first place, as small changes are easier to review even if they are only reviewed by the committer.

Another good idea is to have good commit messages. They can be very helpful to understand why some changes were made.

These general best practices are very helpful if you bisect often.

Avoiding bug prone merges

First merges by themselves can introduce some regressions even when the merge needs no source code conflict resolution. This is because a semantic change can happen in one branch while the other branch is not aware of it.

For example one branch can change the semantic of a function while the other branch add more calls to the same function.

This is made much worse if many files have to be fixed to resolve conflicts. That’s why such merges are called "evil merges". They can make regressions very difficult to track down. It can even be misleading to know the first bad commit if it happens to be such a merge, because people might think that the bug comes from bad conflict resolution when it comes from a semantic change in one branch.

Anyway "git rebase" can be used to linearize history. This can be used either to avoid merging in the first place. Or it can be used to bisect on a linear history instead of the non linear one, as this should give more information in case of a semantic change in one branch.

Merges can be also made simpler by using smaller branches or by using many topic branches instead of only long version related branches.

And testing can be done more often in special integration branches like linux-next for the linux kernel.

Adapting your work-flow

A special work-flow to process regressions can give great results.

Here is an example of a work-flow used by Andreas Ericsson:

- write, in the test suite, a test script that exposes the regression
- use "git bisect run" to find the commit that introduced it
- fix the bug that is often made obvious by the previous step
- commit both the fix and the test script (and if needed more tests)

And here is what Andreas said about this work-flow [5]:

> To give some hard figures, we used to have an average report-to-fix cycle of 142.6 hours (according to our somewhat weird bug-tracker which just measures wall-clock time). Since we moved to Git, we've lowered that to 16.2 hours. Primarily because we can stay on top of the bug fixing now, and because everyone's jockeying to get to fix bugs (we're quite proud of how lazy we are to let Git find the bugs for us). Each new release results in ~40% fewer bugs (almost certainly due to how we now feel about writing tests).

Clearly this work-flow uses the virtuous circle between test suites and "git bisect". In fact it makes it the standard procedure to deal with regression.

In other messages Andreas says that they also use the "best practices" described above: small logical commits, topic branches, no evil merge,... These practices all improve the bisectability of the commit graph, by making it easier and more useful to bisect.

So a good work-flow should be designed around the above points. That is making bisecting easier, more useful and standard.

Involving QA people and if possible end users

One nice about "git bisect" is that it is not only a developer tool. It can effectively be used by QA people or even end users (if they have access to the source code or if they can get access to all the builds).

There was a discussion at one point on the linux kernel mailing list of whether it was ok to always ask end user to bisect, and very good points were made to support the point of view that it is ok.

For example David Miller wrote [6]:

> What people don't get is that this is a situation where the "end node principle" applies. When you have limited resources (here: developers) you don't push the bulk of the burden upon them. Instead you push things out to the resource you have a lot of, the end nodes (here: users), so that the situation actually scales.

This means that it is often "cheaper" if QA people or end users can do it.

What is interesting too is that end users that are reporting bugs (or QA people that reproduced a bug) have access to the environment where the bug happens. So they can often more easily reproduce a regression. And if they can bisect, then more information will be extracted from the environment where the bug happens, which means that it will be easier to understand and then fix the bug.

For open source projects it can be a good way to get more useful contributions from end users, and to introduce them to QA and development activities.

Using complex scripts

In some cases like for kernel development it can be worth developing complex scripts to be able to fully automate bisecting.

Here is what Ingo Molnar says about that [7]:

> i have a fully automated bootup-hang bisection script. It is based on "git-bisect run". I run the script, it builds and boots kernels fully automatically, and when the bootup fails (the script notices that via the serial log, which it continuously watches - or via a timeout, if the system does not come up within 10 minutes it's a "bad" kernel), the script raises my attention via a beep and i power cycle the test box. (yeah, i should make use of a managed power outlet to 100% automate it)

Combining test suites, git bisect and other systems together

We have seen that test suites an git bisect are very powerful when used together. It can be even more powerful if you can combine them with other systems.

For example some test suites could be run automatically at night with some unusual (or even random) configurations. And if a regression is found by a test suite, then "git bisect" can be automatically launched, and its result can be emailed to the author of the first bad commit found by "git bisect", and perhaps other people too. And a new entry in the bug tracking system could be automatically created too.

The future of bisecting

"git replace"

We saw earlier that "git bisect skip" is now using a PRNG to try to avoid areas in the commit graph where commits are untestable. The problem is that sometimes the first bad commit will be in an untestable area.

To simplify the discussion we will suppose that the untestable area is a simple string of commits and that it was created by a breakage introduced by one commit (let's call it BBC for bisect breaking commit) and later fixed by another one (let's call it BFC for bisect fixing commit).

For example:

```
...-Y-BBC-X1-X2-X3-X4-X5-X6-BFC-Z-...
```

where we know that Y is good and BFC is bad, and where BBC and X1 to X6 are untestable.

In this case if you are bisecting manually, what you can do is create a special branch that starts just before the BBC. The first commit in this branch should be the BBC with the BFC squashed into it. And the other commits in the branch should be the commits between BBC and BFC rebased on the first commit of the branch and then the commit after BFC also rebased on.

For example:

```
      (BBC+BFC)-X1'-X2'-X3'-X4'-X5'-X6'-Z'
     /
...-Y-BBC-X1-X2-X3-X4-X5-X6-BFC-Z-...
```

where commits quoted with ' have been rebased.

You can easily create such a branch with Git using interactive rebase.

For example using:

```
$ git rebase -i Y Z
```

and then moving BFC after BBC and squashing it.

After that you can start bisecting as usual in the new branch and you should eventually find the first bad commit.

For example:

```
$ git bisect start Z' Y
```

If you are using "git bisect run", you can use the same manual fix up as above, and then start another "git bisect run" in the special branch. Or as the "git bisect" man page says, the script passed to "git bisect run" can apply a patch before it compiles and test the software [8]. The patch should turn a current untestable commits into a testable one. So the testing will result in "good" or "bad" and "git bisect" will be able to find the first bad commit. And the script should not forget to remove the patch once the testing is done before exiting from the script.

(Note that instead of a patch you can use "git cherry-pick BFC" to apply the fix, and in this case you should use "git reset --hard HEAD^" to revert the cherry-pick after testing and before returning from the script.)

But the above ways to work around untestable areas are a little bit clunky. Using special branches is nice because these branches can be shared by developers like usual branches, but the risk is that people will get many such branches. And it disrupts the normal "git bisect" work-flow. So, if you want to use "git bisect run" completely automatically, you have to add special code in your script to restart bisection in the special branches.

Anyway one can notice in the above special branch example that the Z' and Z commits should point to the same source code state (the same "tree" in git parlance). That's because Z' result from applying the same changes as Z just in a slightly different order.

So if we could just "replace" Z by Z' when we bisect, then we would not need to add anything to a script. It would just work for anyone in the project sharing the special branches and the replacements.

With the example above that would give:

```
      (BBC+BFC)-X1'-X2'-X3'-X4'-X5'-X6'-Z'-...
     /
...-Y-BBC-X1-X2-X3-X4-X5-X6-BFC-Z
```

That's why the "git replace" command was created. Technically it stores replacements "refs" in the "refs/replace/" hierarchy. These "refs" are like branches (that are stored in "refs/heads/") or tags (that are stored in "refs/tags"), and that means that they can automatically be shared like branches or tags among developers.

"git replace" is a very powerful mechanism. It can be used to fix commits in already released history, for example to change the commit message or the author. And it can also be used instead of git "grafts" to link a repository with another old repository.

In fact it's this last feature that "sold" it to the Git community, so it is now in the "master" branch of Git's Git repository and it should be released in Git 1.6.5 in October or November 2009.

One problem with "git replace" is that currently it stores all the replacements refs in "refs/replace/", but it would be perhaps better if the replacement refs that are useful only for bisecting would be in "refs/replace/bisect/". This way the replacement refs could be used only for bisecting, while other refs directly in "refs/replace/" would be used nearly all the time.

Bisecting sporadic bugs

Another possible improvement to "git bisect" would be to optionally add some redundancy to the tests performed so that it would be more reliable when tracking sporadic bugs.

This has been requested by some kernel developers because some bugs called sporadic bugs do not appear in all the kernel builds because they are very dependent on the compiler output.

The idea is that every 3 test for example, "git bisect" could ask the user to test a commit that has already been found to be "good" or "bad" (because one of its descendants or one of its ancestors has been found to be "good" or "bad" respectively). If it happens that a commit has been previously incorrectly classified then the bisection can be aborted early, hopefully before too many mistakes have been made. Then the user will have to look at what happened and then restart the bisection using a fixed bisect log.

There is already a project called BBChop created by Ealdwulf Wuffinga on Github that does something like that using Bayesian Search Theory [9]:

> BBChop is like *git bisect* (or equivalent), but works when your bug is intermittent. That is, it works in the presence of false negatives (when a version happens to work this time even though it contains the bug). It assumes that there are no false positives (in principle, the same approach would work, but adding it may be non-trivial).

But BBChop is independent of any VCS and it would be easier for Git users to have something integrated in Git.

Conclusion

We have seen that regressions are an important problem, and that "git bisect" has nice features that complement very well practices and other tools, especially test suites, that are generally used to fight regressions. But it might be needed to change some work-flows and (bad) habits to get the most out of it.

Some improvements to the algorithms inside "git bisect" are possible and some new features could help in some cases, but overall "git bisect" works already very well, is used a lot, and is already very useful. To back up that last claim, let's give the final word to Ingo Molnar when he was asked by the author how much time does he think "git bisect" saves him when he uses it:

> a *lot*.
>
> About ten years ago did i do my first *bisection* of a Linux patch queue. That was prior the Git (and even prior the BitKeeper) days. I literally days spent sorting out patches, creating what in essence were standalone commits that i guessed to be related to that bug.
>
> It was a tool of absolute last resort. I'd rather spend days looking at printk output than do a manual *patch bisection*.
>
> With Git bisect it's a breeze: in the best case i can get a ~15 step kernel bisection done in 20-30 minutes, in an automated way. Even with manual help or when bisecting multiple, overlapping bugs, it's rarely more than an hour.
>
> In fact it's invaluable because there are bugs i would never even *try* to debug if it wasn't for git bisect. In the past there were bug patterns that were immediately hopeless for me to debug - at best i could send the crash/bug signature to lkml and hope that someone else can think of something.
>
> And even if a bisection fails today it tells us something valuable about the bug: that it's non-deterministic - timing or kernel image layout dependent.
>
> So git bisect is unconditional goodness - and feel free to quote that ;-)

Acknowledgments

Many thanks to Junio Hamano for his help in reviewing this paper, for reviewing the patches I sent to the Git mailing list, for discussing some ideas and helping me improve them, for improving "git bisect" a lot and for his awesome work in maintaining and developing Git.

Many thanks to Ingo Molnar for giving me very useful information that appears in this paper, for commenting on this paper, for his suggestions to improve "git bisect" and for evangelizing "git bisect" on the linux kernel mailing lists.

Many thanks to Linus Torvalds for inventing, developing and evangelizing "git bisect", Git and Linux.

Many thanks to the many other great people who helped one way or another when I worked on Git, especially to Andreas Ericsson, Johannes Schindelin, H. Peter Anvin, Daniel Barkalow, Bill Lear, John Hawley, Shawn O. Pierce, Jeff King, Sam Vilain, Jon Seymour.

Many thanks to the Linux-Kongress program committee for choosing the author to given a talk and for publishing this paper.

References

- [1] *Software Errors Cost U.S. Economy $59.5 Billion Annually*. Nist News Release.
- [2] *Code Conventions for the Java Programming Language*. Sun Microsystems.
- [3] *Software maintenance*. Wikipedia.
- [4] Junio C Hamano. *Automated bisect success story*. Gmane.
- [5] Christian Couder. *Fully automated bisecting with "git bisect run"*. LWN.net.
- [6] Jonathan Corbet. *Bisection divides users and developers*. LWN.net.
- [7] Ingo Molnar. *Re: BUG 2.6.23-rc3 can't see sd partitions on Alpha*. Gmane.
- [8] Junio C Hamano and the git-list. *git-bisect(1) Manual Page*. Linux Kernel Archives.
- [9] Ealdwulf. *bbchop*. GitHub.

Last updated 2014-11-27 19:58:07 CET

gitcli(7) Manual Page

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:

- 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 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 asking 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.

  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 checkout -- *.c
  $ git checkout -- \*.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:

- it’s preferred to use the non-dashed form of Git commands, which means that you should prefer `git foo` to `git-foo`.
- 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.

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
```

--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.

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`.

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`.

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.

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 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 http://marc.info/?l=git&m=116563135620359 and http://marc.info/?l=git&m=119150393620273 for further information.

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:58:08 CET

gitcore-tutorial(7) Manual Page

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" (gittutorial(7)) or the Git User Manual.

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 for 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 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 rsync://rsync.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 checkout -b 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 checkout* what the base of the checkout would be. In other words, if you have an earlier tag or branch, you'd just do

```
$ git checkout -b 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 checkout 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 checkout* 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 checkout 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 checkout 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 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 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 checkout 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:

Rsync

`rsync://remote.machine/path/to/repo.git/`

Rsync transport is usable for both uploading and downloading, but is completely unaware of what git does, and can produce unexpected results when you download from the public repository while the repository owner is uploading into it via `rsync` transport. Most notably, it could update the files under `refs/` which holds the object name of the topmost commits before uploading the files in `objects/` — the downloader would obtain head commit object name while that object itself is still not available in the repository. For this reason, it is considered deprecated.

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 http://www.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 checkout mybranch
$ git reset --hard master^2
$ git checkout 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 `.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, unless rsync transport is used. 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.

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 *git clone* 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 CVS background, the style of cooperation suggested in the previous section may be new to you. You do not have to worry. Git supports "shared public repository" style of cooperation you are probably more familiar with as well.

See 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 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

gittutorial(7), gittutorial-2(7), gitcvs-migration(7), git-help(1), giteveryday(7), The Git User's Manual

GIT

Part of the git(1) suite.

Last updated 2014-11-27 19:58:08 CET

gitcredentials(7) Manual Page

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. 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.

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 git-credential-cache(1) for details.

store
: Store credentials indefinitely on disk. See 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
```

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 password, no more helpers will be tried.

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.

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.

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. See the documentation for Git's credentials API for details.

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

gitcvs-migration(7) Manual Page

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 gittutorial(7) and 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 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 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 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

First, install version 2.1 or higher of cvsps from http://www.cobite.com/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 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 githooks(5).

You can enforce finer grained permissions using update hooks. See Controlling access to branches using update hooks.

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 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

gittutorial(7), gittutorial-2(7), gitcore-tutorial(7), gitglossary(7), giteveryday(7), The Git User's Manual

GIT

Part of the git(1) suite.

Last updated 2014-11-27 19:58:08 CET

gitdiffcore(7) Manual Page

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

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 Detection 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. 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 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 a 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 old version prefixed with `-`, followed by the entire contents of 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.

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
```

SEE ALSO

git-diff(1), git-diff-files(1), git-diff-index(1), git-diff-tree(1), git-format-patch(1), git-log(1), gitglossary(7), The Git User's Manual

GIT

Part of the git(1) suite.

Last updated 2014-11-27 19:56:10 CET

gitglossary(7) Manual Page

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 an active 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 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.

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 these cases, you do not make a new merge commit but instead just update to his revision. 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 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.

grafts
: Grafts enables 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 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 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 can not 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
: 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.

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`.

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 worktree. 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 a 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.

exclude
: After a path matches any non-exclude pathspec, it will be run through all exclude pathspec (magic signature: `!`). If it matches, the path is ignored.

parent
: A commit object contains a (possibly empty) list of the logical predecessor(s) in the line of development, i.e. its parents.

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 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.

pull
: Pulling a branch means to fetch it and merge it. See also git-pull(1).

push
: Pushing a branch means to get the branch's head ref from a remote repository, find out if it is a direct 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.

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 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 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-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 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 git-clone(1), and its history can be later deepened with git-fetch(1).

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 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.

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.

SEE ALSO

gittutorial(7), gittutorial-2(7), gitcvs-migration(7), giteveryday(7), The Git User's Manual

GIT

Part of the git(1) suite.

Last updated 2014-11-27 19:58:08 CET

githooks(5) Manual Page

NAME

githooks - Hooks used by Git

SYNOPSIS

$GIT_DIR/hooks/*

DESCRIPTION

Hooks are little scripts you can place in `$GIT_DIR/hooks` directory to trigger action at certain points. When *git init* is run, a handful of example hooks are copied into the `hooks` directory of the new repository, but by default they are all disabled. To enable a hook, rename it by removing its `.sample` suffix.

Note It is also a requirement for a given hook to be executable. However - in a freshly initialized repository - the `.sample` files are executable by default.

This document describes the currently defined hooks.

HOOKS

applypatch-msg

This hook is invoked by *git am* script. It takes a single parameter, the name of the file that holds the proposed commit log message. Exiting with 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 (if the project has one). 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.

pre-applypatch

This hook is invoked by *git am*. 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 test.

The default *pre-applypatch* hook, when enabled, runs the *pre-commit* hook, if the latter is enabled.

post-applypatch

This hook is invoked by *git am*. 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*.

pre-commit

This hook is invoked by *git commit*, and can be bypassed with `--no-verify` option. It takes no parameter, and is invoked before obtaining the proposed commit log message and making a commit. Exiting with non-zero status from this script causes the *git commit* to abort.

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.

prepare-commit-msg

This hook is invoked by *git commit* 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 SHA-1 (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 replacement for pre-commit hook.

The sample `prepare-commit-msg` hook that comes with Git comments out the `Conflicts:` part of a merge's commit message.

commit-msg

This hook is invoked by *git commit*, and can be bypassed with `--no-verify` option. It takes a single parameter, the name of the file that holds the proposed commit log message. Exiting with non-zero status causes the *git commit* 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 (if the project has one). 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" lines, and aborts the commit if one is found.

post-commit

This hook is invoked by *git commit*. It takes no parameter, and is invoked after a commit is made.

This hook is meant primarily for notification, and cannot affect the outcome of *git commit*.

pre-rebase

This hook is called by *git rebase* 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.

post-checkout

This hook is invoked when a *git checkout* 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 checkout*.

It is also run after *git clone*, 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.

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.

post-merge

This hook is invoked by *git merge*, 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/setgitperms.perl for an example of how to do this.

pre-push

This hook is called by *git push* 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 sha1> SP <remote ref> SP <remote sha1> 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, 40-character SHA-1s would be supplied. If the foreign ref does not yet exist the `<remote SHA-1>` will be 40 `0`. If a ref is to be deleted, the `<local ref>` will be supplied as `(delete)` and the `<local SHA-1>` will be 40 `0`. If the local commit was specified by something other than a name which could be expanded (such as `HEAD~`, or a SHA-1) 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.

pre-receive

This hook is invoked by *git-receive-pack* on the remote repository, which happens when a *git push* is done on a local 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-value> SP <new-value> SP <ref-name> LF
```

where `<old-value>` is the old object name stored in the ref, `<new-value>` 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-value>` is 40 `0`.

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.

update

This hook is invoked by *git-receive-pack* on the remote repository, which happens when a *git push* is done on a local 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.

Another use suggested on the mailing list is to use this hook to implement access control which is finer grained than the one based on filesystem group.

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 to be pushed.

post-receive

This hook is invoked by *git-receive-pack* on the remote repository, which happens when a *git push* is done on a local 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.

post-update

This hook is invoked by *git-receive-pack* on the remote repository, which happens when a *git push* is done on a local 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.

push-to-checkout

This hook is invoked by *git-receive-pack* on the remote repository, which happens when a *git push* is done on a local repository, 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 `read-tree -u -m` is essentially the same as `git checkout` that switches branches while keeping the local changes in the working tree that do not interfere with the difference between the branches.

pre-auto-gc

This hook is invoked by *git gc --auto*. It takes no parameter, and exiting with non-zero status from this script causes the *git gc --auto* to abort.

post-rewrite

This hook is invoked by commands that rewrite commits (`git commit --amend`, *git-rebase*; currently *git-filter-branch* does *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-sha1> SP <new-sha1> [ 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 git-config.txt(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-sha1*.

  The commits are guaranteed to be listed in the order that they were processed by rebase.

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

gitignore(5) Manual Page

NAME

gitignore - Specifies intentionally untracked files to ignore

SYNOPSIS

$HOME/.config/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, with patterns in the higher level files (up to the toplevel of the work tree) 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`".
- If the pattern ends with a slash, it is removed for the purpose of the following description, but it would only find a match with a directory. In other words, `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).
- If the pattern does not contain a slash `/`, Git treats it as a shell glob pattern and checks for a match against the pathname relative to the location of the `.gitignore` file (relative to the toplevel of the work tree if not from a `.gitignore` file).
- Otherwise, 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".
- A leading slash matches the beginning of the pathname. For example, "/*.c" matches "cat-file.c" but not "mozilla-sha1/sha1.c".

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.

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*.

EXAMPLES

```
    $ 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

git-rm(1), gitrepository-layout(5), git-check-ignore(1)

GIT

Part of the git(1) suite

Last updated 2015-03-26 21:44:44 CET

gitmodules(5) Manual Page

NAME

gitmodules - defining submodule properties

SYNOPSIS

$GIT_WORK_DIR/.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 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 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 "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*. See description of *update* command in git-submodule(1) for their meaning. Note that the *!command* form is intentionally ignored here for security reasons.

submodule.<name>.branch
: A remote branch name for tracking updates in the upstream submodule. If the option is not specified, it defaults to *master*. See the `--remote` documentation in 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 submodules 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. When set to "all", it will never be considered modified (but will nonetheless show up in the output of status and commit when it has been staged), "dirty" will ignore all changes to the submodules 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. If this option is also present in the submodules 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-submodule" option. The *git submodule* commands are not affected by this setting.

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

git-submodule(1) git-config(1)

GIT

Part of the git(1) suite

gitnamespaces(7) Manual Page

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 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 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.

git-upload-pack(1) and 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, git-http-backend(1), will pass GIT_NAMESPACE through to the backend programs; see git-http-backend(1) for sample configuration to expose repository namespaces as repositories.

For a simple local test, you can use git-remote-ext(1):

```
git clone ext::'git --namespace=foo %s /tmp/prefixed.git'
```

SECURITY

Anyone with access to any namespace within a repository can potentially access objects from any other namespace stored in the same repository. You can't directly say "give me object ABCD" if you don't have a ref to it, but you can do some other sneaky things like:

1. Claiming to push ABCD, at which point the server will optimize out the need for you to actually send it. Now you have a ref to ABCD and can fetch it (claiming not to have it, of course).
2. Requesting other refs, claiming that you have ABCD, at which point the server may generate deltas against ABCD.

None of this causes a problem if you only host public repositories, or if everyone who may read one namespace may also read everything in every other namespace (for instance, if everyone in an organization has read permission to every repository).

Last updated 2014-11-27 19:54:14 CET

gitremote-helpers(1) Manual Page

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.

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.

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*.

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*.

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*.

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.

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.

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 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 git-fast-export(1).

signed-tags
: This modifies the *export* capability, instructing Git to pass *--signed-tags=verbatim* to git-fast-export(1). In the absence of this capability, Git will use *--signed-tags=warn-strip*.

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, 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.

  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 a file under GIT_DIR/objects/pack which is keeping a pack until refs can be suitably updated.

  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.

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.

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 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.

SEE ALSO

git-remote(1)

git-remote-ext(1)

git-remote-fd(1)

git-remote-testgit(1)

git-fast-import(1)

GIT

Part of the git(1) suite

Last updated 2014-12-13 19:39:10 CET

gitrepository-layout(5) Manual Page

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 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 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.

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 object 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.

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 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 git-pack-refs(1).

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 git-checkout(1) for details.

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 git-fetch(1) for details. This mechanism is legacy and not likely to be found in modern repositories.

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 githooks(5) for more details about each hook.

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.

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 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: gitignore(5).

info/sparse-checkout
: This file stores sparse checkout patterns. See also: 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 git-fetch(1) for details. This mechanism is legacy and not likely to be found in modern repositories.

logs
: Records of changes made to refs are stored in this directory. See git-update-ref(1) for more information.

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 git-clone(1) and git-fetch(1).

modules
: Contains the git-repositories of the submodules.

SEE ALSO

git-init(1), git-clone(1), git-fetch(1), git-pack-refs(1), git-gc(1), git-checkout(1), gitglossary(7), The Git User's Manual

GIT

Part of the git(1) suite.

Last updated 2014-11-27 19:58:08 CET

gitrevisions(7) Manual Page

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 git-log(1)), all commits which can be reached from that commit. In the latter case one can also specify a range of revisions explicitly.

In addition, some Git commands (such as git-show(1)) 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.

<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* and *CHERRY_PICK_HEAD*);
    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, 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`. *CHERRY_PICK_HEAD* records the commit which you are cherry-picking when you run `git cherry-pick`.

       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}*
: The suffix *@{upstream}* to a branchname (short form *<branchname>@{u}*) refers to the branch that the branch specified by branchname is set to build on top of (configured with `branch.<name>.remote` and `branch.<name>.merge`). A missing branchname defaults to the current one.

*<rev>^*, 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. *master~3*
: 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. If the commit message starts with a *!* you have to repeat that; the special sequence *:/!*, followed by something else than *!*, is reserved for now. The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. *:/^foo*.

<rev>:<path>, e.g. *HEAD:README, :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. *:path* (with an empty part before the colon) is a special case of the syntax described next: content recorded in the index at the given path. 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  = 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. To these commands, specifying a single revision with the notation described in the previous section means the set of commits reachable from that commit, following the commit ancestry chain.

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*.

This 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*.

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* or *r2* but not from both.

In these two shorthands, 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.

Two other shorthands for naming a set that is formed by a commit and its parent commits exist. The *r1^@* notation means all parents of *r1*. *r1^!* includes commit *r1* but excludes all of its parents.

To summarize:

<rev>
: Include commits that are reachable from (i.e. ancestors of) <rev>.

^<rev>
: Exclude commits that are reachable from (i.e. ancestors of) <rev>.

<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 then all its parents prefixed with ^ to exclude them (and their ancestors).

Here are a handful of examples:

```
D                G H D
D F              G H I J D F
^G D             H D
^D B             E I J F B
B..C             C
B...C            G H D E B C
^D B C           E I J F B C
C                I J F C
C^@              I J F
C^!              C
F^! D            G H D F
```

SEE ALSO

git-rev-parse(1)

GIT

Part of the git(1) suite

A short Git tools survey

Introduction

Apart from Git contrib/ area there are some others third-party tools you may want to look.

This document presents a brief summary of each tool and the corresponding link.

Alternative/Augmentative Porcelains

- **Cogito** (http://www.kernel.org/pub/software/scm/cogito/)

  ```
  Cogito is a version control system layered on top of the Git tree history
  storage system. It aims at seamless user interface and ease of use,
  providing generally smoother user experience than the "raw" Core Git
  itself and indeed many other version control systems.
  ```

  ```
  Cogito is no longer maintained as most of its functionality
  is now in core Git.
  ```

- **pg** (http://www.spearce.org/category/projects/scm/pg/)

  ```
  pg is a shell script wrapper around Git to help the user manage a set of
  patches to files. pg is somewhat like quilt or StGit, but it does have a
  slightly different feature set.
  ```

- **StGit** (http://www.procode.org/stgit/)

  ```
  Stacked Git provides a quilt-like patch management functionality in the
  Git environment. You can easily manage your patches in the scope of Git
  until they get merged upstream.
  ```

History Viewers

- **gitk** (shipped with git-core)

  ```
  gitk is a simple Tk GUI for browsing history of Git repositories easily.
  ```

- **gitview** (contrib/)

  ```
  gitview is a GTK based repository browser for Git
  ```

- **gitweb** (shipped with git-core)

  ```
  Gitweb provides full-fledged web interface for Git repositories.
  ```

- **qgit** (http://digilander.libero.it/mcostalba/)

  ```
  QGit is a git/StGit GUI viewer built on Qt/C++. QGit could be used
  to browse history and directory tree, view annotated files, commit
  changes cherry picking single files or applying patches.
  Currently it is the fastest and most feature rich among the Git
  viewers and commit tools.
  ```

- **tig** (http://jonas.nitro.dk/tig/)

  ```
  tig by Jonas Fonseca is a simple Git repository browser
  written using ncurses. Basically, it just acts as a front-end
  for git-log and git-show/git-diff. Additionally, you can also
  use it as a pager for Git commands.
  ```

Foreign SCM interface

- **git-svn** (shipped with git-core)

  ```
  git-svn is a simple conduit for changesets between a single Subversion
  branch and Git.
  ```

- **quilt2git / git2quilt** (http://home-tj.org/wiki/index.php/Misc)

```
These utilities convert patch series in a quilt repository and commit
series in Git back and forth.
```

- **hg-to-git** (contrib/)

```
hg-to-git converts a Mercurial repository into a Git one, and
preserves the full branch history in the process. hg-to-git can
also be used in an incremental way to keep the Git repository
in sync with the master Mercurial repository.
```

Others

- **(h)gct** (http://www.cyd.liu.se/users/~freku045/gct/)

```
Commit Tool or (h)gct is a GUI enabled commit tool for Git and
Mercurial (hg). It allows the user to view diffs, select which files
to committed (or ignored / reverted) write commit messages and
perform the commit itself.
```

- **git.el** (contrib/)

```
This is an Emacs interface for Git. The user interface is modelled on
pcl-cvs. It has been developed on Emacs 21 and will probably need some
tweaking to work on XEmacs.
```

http://git.or.cz/gitwiki/InterfacesFrontendsAndTools has more comprehensive list.

Last updated 2014-11-27 19:55:05 CET

gitweb.conf(5) Manual Page

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 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 exists 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.

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 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 gitweb(1) manpage.

$strict_export
: Only allow viewing of repositories also shown on the overview page. This for example makes `$gitweb_export_ok` file decide if repository is available and not only if it is shown. If `$gitweb_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).

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://www.andre-simon.de 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: if you want to add support for new file type (supported by "highlight" but not used by gitweb), you need to modify `%highlight_ext` or `%highlight_basename`, depending on whether you detect type of file based on extension (for example "sh") or on its basename (for example "Makefile"). The keys of these hashes are extension and basename, respectively, and value for given key is name of syntax to be passed via `--syntax <syntax>` to highlighter.

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';
```

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, http://git-scm.com; in the past, they pointed to Git documentation at http://www.kernel.org.

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.

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').

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.

$projects_list_group_categories
: Whether to enables the grouping of projects by category on the project list page. The category of a project is determined by the `$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.

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.

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 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 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.blame` 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 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 git-log(1) (the description of `-S<string>` option, which refers to pickaxe entry in 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 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 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.

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. 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 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 http://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

gitweb(1), git-instaweb(1)

gitweb/README, *gitweb/INSTALL*

GIT

Part of the git(1) suite

Last updated 2015-05-03 21:16:24 CEST

gitweb(1) Manual Page

NAME

gitweb - Git web interface (web frontend to Git repositories)

SYNOPSIS

To get started with gitweb, run git-instaweb(1) from a Git repository. This would configure and start your web server, and run 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, see what was changed when, by who.
- 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 step through revisions one at a time, viewing the history of the repository.
- Finding commits which commit messages matches given search term.

See http://git.kernel.org/?p=git/git.git;a=tree;f=gitweb or http://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 gitweb.conf(5) for details.

Repositories

Gitweb can show information from one or more Git repositories. These repositories have to be all on local filesystem, and have to share common repository root, i.e. be all under a single parent repository (but see also "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 `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".

Projects list file format

Instead of having gitweb find repositories by scanning 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 http://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 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).

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.

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 `$export_ok` variable in gitweb config file; see gitweb.conf(5) manpage. If it evaluates to true, gitweb shows repositories only if this file named by `$export_ok` exists in its object database (if directory has the magic file named `$export_ok`).

For example 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;
};
```

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 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 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 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 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 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 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];
```

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 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.

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
```

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
```

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).

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 browseable gitweb interface. If you then start your 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 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.

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 folder 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.

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, putting "gitweb" in the subject of email.

SEE ALSO

gitweb.conf(5), git-instaweb(1)

gitweb/README, *gitweb/INSTALL*

GIT

Part of the git(1) suite

Last updated 2014-11-27 19:57:04 CET

Git Howto Index

Here is a collection of mailing list postings made by various people describing how they use Git in their workflow.

- keep-canonical-history-correct by Junio C Hamano <gitster@pobox.com>

This how-to explains a method for keeping a project's history correct when using git pull.

- maintain-git by Junio C Hamano <gitster@pobox.com>

Imagine that Git development is racing along as usual, when our friendly neighborhood maintainer is struck down by a wayward bus. Out of the hordes of suckers (loyal developers), you have been tricked (chosen) to step up as the new maintainer. This howto will show you "how to" do it.

- new-command by Eric S. Raymond <esr@thyrsus.com>

This is how-to documentation for people who want to add extension commands to Git. It should be read alongside api-builtin.txt.

- rebase-from-internal-branch by Junio C Hamano <gitster@pobox.com>

In this article, JC talks about how he rebases the public "pu" branch using the core Git tools when he updates the "master" branch, and how "rebase" works. Also discussed is how this applies to individual developers who sends patches upstream.

- rebuild-from-update-hook by Junio C Hamano <gitster@pobox.com>

In this how-to article, JC talks about how he uses the post-update hook to automate Git documentation page shown at http://www.kernel.org/pub/software/scm/git/docs/.

- recover-corrupted-blob-object by Linus Torvalds <torvalds@linux-foundation.org>

Some tricks to reconstruct blob objects in order to fix a corrupted repository.

- recover-corrupted-object-harder by Jeff King <peff@peff.net>

Recovering a corrupted object when no good copy is available.

- revert-a-faulty-merge by Linus Torvalds <torvalds@linux-foundation.org>, Junio C Hamano <gitster@pobox.com>

Sometimes a branch that was already merged to the mainline is later found to be faulty. Linus and Junio give guidance on recovering from such a premature merge and continuing development after the offending branch is fixed.

- revert-branch-rebase by Junio C Hamano <gitster@pobox.com>

In this article, JC gives a small real-life example of using *git revert* command, and using a temporary branch and tag for safety and easier sanity checking.

- separating-topic-branches by Junio C Hamano <gitster@pobox.com>

In this article, JC describes how to separate topic branches.

- setup-git-server-over-http by Rutger Nijlunsing <rutger@nospam.com>
- update-hook-example by Junio C Hamano <gitster@pobox.com> and Carl Baldwin <cnb@fc.hp.com>

An example hooks/update script is presented to implement repository maintenance policies, such as who can push into which branch and who can make a tag.

- use-git-daemon
- using-merge-subtree by Sean <seanlkml@sympatico.ca>

In this article, Sean demonstrates how one can use the subtree merge strategy.

- using-signed-tag-in-pull-request by Junio C Hamano <gitster@pobox.com>

Beginning v1.7.9, a contributor can push a signed tag to her publishing repository and ask her integrator to pull it. This assures the integrator that the pulled history is authentic and allows others to later validate it.

Last updated 2015-05-03 21:25:49 CEST

Keep authoritative canonical history correct with git pull

Sometimes a new project integrator will end up with project history that appears to be "backwards" from what other project developers expect. This howto presents a suggested integration workflow for maintaining a central repository.

Suppose that that central repository has this history:

```
---o---o---A
```

which ends at commit `A` (time flows from left to right and each node in the graph is a commit, lines between them indicating parent-child relationship).

Then you clone it and work on your own commits, which leads you to have this history in **your** repository:

```
---o---o---A---B---C
```

Imagine your coworker did the same and built on top of `A` in **his** repository in the meantime, and then pushed it to the central repository:

```
---o---o---A---X---Y---Z
```

Now, if you `git push` at this point, because your history that leads to `C` lacks `X`, `Y` and `Z`, it will fail. You need to somehow make the tip of your history a descendant of `Z`.

One suggested way to solve the problem is "fetch and then merge", aka `git pull`. When you fetch, your repository will have a history like this:

```
---o---o---A---B---C
            \
             X---Y---Z
```

Once you run merge after that, while still on **your** branch, i.e. `C`, you will create a merge `M` and make the history look like this:

```
---o---o---A---B---C---M
            \         /
             X---Y---Z
```

`M` is a descendant of `Z`, so you can push to update the central repository. Such a merge `M` does not lose any commit in both histories, so in that sense it may not be wrong, but when people want to talk about "the authoritative canonical history that is shared among the project participants", i.e. "the trunk", they often view it as "commits you see by following the first-parent chain", and use this command to view it:

```
$ git log --first-parent
```

For all other people who observed the central repository after your coworker pushed `Z` but before you pushed `M`, the commit on the trunk used to be `o-o-A-X-Y-Z`. But because you made `M` while you were on `C`, `M`'s first parent is `C`, so by pushing `M` to advance the central repository, you made `X-Y-Z` a side branch, not on the trunk.

You would rather want to have a history of this shape:

```
---o---o---A---X---Y---Z---M'
            \             /
             B-----------C
```

so that in the first-parent chain, it is clear that the project first did `X` and then `Y` and then `Z` and merged a change that consists of two commits `B` and `C` that achieves a single goal. You may have worked on fixing the bug #12345 with these two patches, and the merge `M'` with swapped parents can say in its log message "Merge fix-bug-12345". Having a way to tell `git pull` to create a merge but record the parents in reverse order may be a way to do so.

Note that I said "achieves a single goal" above, because this is important. "Swapping the merge order" only covers a special case where the project does not care too much about having unrelated things done on a single merge but cares a lot about first-parent chain.

There are multiple schools of thought about the "trunk" management.

1. Some projects want to keep a completely linear history without any merges. Obviously, swapping the merge order would not match their taste. You would need to flatten your history on top of the updated upstream to result in a history of this shape instead:

   ```
   ---o---o---A---X---Y---Z---B---C
   ```

with `git pull --rebase` or something.

2. Some projects tolerate merges in their history, but do not worry too much about the first-parent order, and allow fast-forward merges. To them, swapping the merge order does not hurt, but it is unnecessary.
3. Some projects want each commit on the "trunk" to do one single thing. The output of `git log --first-parent` in such a project would show either a merge of a side branch that completes a single theme, or a single commit that completes a single theme by itself. If your two commits `B` and `C` (or they may even be two groups of commits) were solving two independent issues, then the merge `M'` we made in the earlier example by swapping the merge order is still not up to the project standard. It merges two unrelated efforts `B` and `C` at the same time.

For projects in the last category (Git itself is one of them), individual developers would want to prepare a history more like this:

```
                 C0--C1--C2     topic-c
                /
     ---o---o---A               master
                \
                 B0--B1--B2     topic-b
```

That is, keeping separate topics on separate branches, perhaps like so:

```
     $ git clone $URL work && cd work
     $ git checkout -b topic-b master
     $ ... work to create B0, B1 and B2 to complete one theme
     $ git checkout -b topic-c master
     $ ... same for the theme of topic-c
```

And then

```
     $ git checkout master
     $ git pull --ff-only
```

would grab `X`, `Y` and `Z` from the upstream and advance your master branch:

```
                 C0--C1--C2     topic-c
                /
     ---o---o---A---X---Y---Z    master
                \
                 B0--B1--B2     topic-b
```

And then you would merge these two branches separately:

```
     $ git merge topic-b
     $ git merge topic-c
```

to result in

```
                 C0--C1---------C2
                /                 \
     ---o---o---A---X---Y---Z---M---N
                \             /
                 B0--B1-----B2
```

and push it back to the central repository.

It is very much possible that while you are merging topic-b and topic-c, somebody again advanced the history in the central repository to put `W` on top of `Z`, and make your `git push` fail.

In such a case, you would rewind to discard `M` and `N`, update the tip of your *master* again and redo the two merges:

```
     $ git reset --hard origin/master
     $ git pull --ff-only
     $ git merge topic-b
     $ git merge topic-c
```

The procedure will result in a history that looks like this:

```
                 C0--C1--------------C2
                /                     \
     ---o---o---A---X---Y---Z---W---M'--N'
                \                 /
                 B0--B1---------B2
```

See also http://git-blame.blogspot.com/2013/09/fun-with-first-parent-history.html

Last updated 2015-05-03 21:26:08 CEST

How to maintain Git

Activities

The maintainer's Git time is spent on three activities.

- Communication (45%)

  ```
  Mailing list discussions on general design, fielding user
  questions, diagnosing bug reports; reviewing, commenting on,
  suggesting alternatives to, and rejecting patches.
  ```

- Integration (50%)

  ```
  Applying new patches from the contributors while spotting and
  correcting minor mistakes, shuffling the integration and
  testing branches, pushing the results out, cutting the
  releases, and making announcements.
  ```

- Own development (5%)

  ```
  Scratching my own itch and sending proposed patch series out.
  ```

The Policy

The policy on Integration is informally mentioned in "A Note from the maintainer" message, which is periodically posted to this mailing list after each feature release is made.

- Feature releases are numbered as vX.Y.0 and are meant to contain bugfixes and enhancements in any area, including functionality, performance and usability, without regression.
- One release cycle for a feature release is expected to last for eight to ten weeks.
- Maintenance releases are numbered as vX.Y.Z and are meant to contain only bugfixes for the corresponding vX.Y.0 feature release and earlier maintenance releases vX.Y.W (W < Z).
- *master* branch is used to prepare for the next feature release. In other words, at some point, the tip of *master* branch is tagged with vX.Y.0.
- *maint* branch is used to prepare for the next maintenance release. After the feature release vX.Y.0 is made, the tip of *maint* branch is set to that release, and bugfixes will accumulate on the branch, and at some point, the tip of the branch is tagged with vX.Y.1, vX.Y.2, and so on.
- *next* branch is used to publish changes (both enhancements and fixes) that (1) have worthwhile goal, (2) are in a fairly good shape suitable for everyday use, (3) but have not yet demonstrated to be regression free. New changes are tested in *next* before merged to *master*.
- *pu* branch is used to publish other proposed changes that do not yet pass the criteria set for *next*.
- The tips of *master* and *maint* branches will not be rewound to allow people to build their own customization on top of them. Early in a new development cycle, *next* is rewound to the tip of *master* once, but otherwise it will not be rewound until the end of the cycle.
- Usually *master* contains all of *maint* and *next* contains all of *master*. *pu* contains all the topics merged to *next*, but is rebuilt directly on *master*.
- The tip of *master* is meant to be more stable than any tagged releases, and the users are encouraged to follow it.
- The *next* branch is where new action takes place, and the users are encouraged to test it so that regressions and bugs are found before new topics are merged to *master*.

Note that before v1.9.0 release, the version numbers used to be structured slightly differently. vX.Y.Z were feature releases while vX.Y.Z.W were maintenance releases for vX.Y.Z.

A Typical Git Day

A typical Git day for the maintainer implements the above policy by doing the following:

- Scan mailing list. Respond with review comments, suggestions etc. Kibitz. Collect potentially usable patches from the mailing list. Patches about a single topic go to one mailbox (I read my mail in Gnus, and type \C-o to save/append messages in files in mbox format).
- Write his own patches to address issues raised on the list but nobody has stepped up solving. Send it out just like other contributors do, and pick them up just like patches from other contributors (see above).
- Review the patches in the saved mailboxes. Edit proposed log message for typofixes and clarifications, and add Acks collected from the list. Edit patch to incorporate "Oops, that should have been like this" fixes from the discussion.
- Classify the collected patches and handle *master* and *maint* updates:
- Obviously correct fixes that pertain to the tip of *maint* are directly applied to *maint*.
- Obviously correct fixes that pertain to the tip of *master* are directly applied to *master*.
- Other topics are not handled in this step.

```
This step is done with "git am".

$ git checkout master    ;# or "git checkout maint"
$ git am -sc3 mailbox
$ make test

In practice, almost no patch directly goes to 'master' or
'maint'.
```

- Review the last issue of "What's cooking" message, review the topics ready for merging (topic→master and topic→maint). Use "Meta/cook -w" script (where Meta/ contains a checkout of the *todo* branch) to aid this step.

```
And perform the merge.  Use "Meta/Reintegrate -e" script (see
later) to aid this step.

$ Meta/cook -w last-issue-of-whats-cooking.mbox

$ git checkout master    ;# or "git checkout maint"
$ echo ai/topic | Meta/Reintegrate -e ;# "git merge ai/topic"
$ git log -p ORIG_HEAD.. ;# final review
$ git diff ORIG_HEAD..   ;# final review
$ make test              ;# final review
```

- Handle the remaining patches:
- Anything unobvious that is applicable to *master* (in other words, does not depend on anything that is still in *next* and not in *master*) is applied to a new topic branch that is forked from the tip of *master*. This includes both enhancements and unobvious fixes to *master*. A topic branch is named as ai/topic where "ai" is two-letter string named after author's initial and "topic" is a descriptive name of the topic (in other words, "what's the series is about").
- An unobvious fix meant for *maint* is applied to a new topic branch that is forked from the tip of *maint*. The topic is named as ai/maint-topic.
- Changes that pertain to an existing topic are applied to the branch, but:
- obviously correct ones are applied first;
- questionable ones are discarded or applied to near the tip;
- Replacement patches to an existing topic are accepted only for commits not in *next*.

```
The above except the "replacement" are all done with:

$ git checkout ai/topic ;# or "git checkout -b ai/topic master"
$ git am -sc3 mailbox

while patch replacement is often done by:

$ git format-patch ai/topic~$n..ai/topic ;# export existing

then replace some parts with the new patch, and reapplying:

$ git checkout ai/topic
$ git reset --hard ai/topic~$n
$ git am -sc3 -s 000*.txt

The full test suite is always run for 'maint' and 'master'
after patch application; for topic branches the tests are run
as time permits.
```

- Merge maint to master as needed:

```
$ git checkout master
$ git merge maint
$ make test
```

- Merge master to next as needed:

```
$ git checkout next
$ git merge master
$ make test
```

- Review the last issue of "What's cooking" again and see if topics that are ready to be merged to *next* are still in good shape (e.g. has there any new issue identified on the list with the series?)
- Prepare *jch* branch, which is used to represent somewhere between *master* and *pu* and often is slightly ahead of *next*.

```
$ Meta/Reintegrate master..pu >Meta/redo-jch.sh
```

```
The result is a script that lists topics to be merged in order to
rebuild 'pu' as the input to Meta/Reintegrate script.  Remove
later topics that should not be in 'jch' yet.  Add a line that
consists of '### match next' before the name of the first topic
in the output that should be in 'jch' but not in 'next' yet.
```

- Now we are ready to start merging topics to *next*. For each branch whose tip is not merged to *next*, one of three things can happen:
- The commits are all next-worthy; merge the topic to next;
- The new parts are of mixed quality, but earlier ones are next-worthy; merge the early parts to next;
- Nothing is next-worthy; do not do anything.

```
This step is aided with Meta/redo-jch.sh script created earlier.
If a topic that was already in 'next' gained a patch, the script
would list it as "ai/topic~1".  To include the new patch to the
updated 'next', drop the "~1" part; to keep it excluded, do not
touch the line.  If a topic that was not in 'next' should be
merged to 'next', add it at the end of the list.  Then:
```

```
$ git checkout -B jch master
$ Meta/redo-jch.sh -c1
```

```
to rebuild the 'jch' branch from scratch.  "-c1" tells the script
to stop merging at the first line that begins with '###'
(i.e. the "### match next" line you added earlier).
```

```
At this point, build-test the result.  It may reveal semantic
conflicts (e.g. a topic renamed a variable, another added a new
reference to the variable under its old name), in which case
prepare an appropriate merge-fix first (see appendix), and
rebuild the 'jch' branch from scratch, starting at the tip of
'master'.
```

```
Then do the same to 'next'
```

```
$ git checkout next
$ sh Meta/redo-jch.sh -c1 -e
```

```
The "-e" option allows the merge message that comes from the
history of the topic and the comments in the "What's cooking" to
be edited.  The resulting tree should match 'jch' as the same set
of topics are merged on 'master'; otherwise there is a mismerge.
Investigate why and do not proceed until the mismerge is found
and rectified.
```

```
$ git diff jch next
```

```
When all is well, clean up the redo-jch.sh script with
```

```
$ sh Meta/redo-jch.sh -u
```

```
This removes topics listed in the script that have already been
merged to 'master'.  This may lose '### match next' marker;
add it again to the appropriate place when it happens.
```

- Rebuild *pu*.

```
$ Meta/Reintegrate master..pu >Meta/redo-pu.sh
```

```
Edit the result by adding new topics that are not still in 'pu'
in the script.  Then
```

```
$ git checkout -B pu jch
$ sh Meta/redo-pu.sh

When all is well, clean up the redo-pu.sh script with

$ sh Meta/redo-pu.sh -u

Double check by running

$ git branch --no-merged pu

to see there is no unexpected leftover topics.

At this point, build-test the result for semantic conflicts, and
if there are, prepare an appropriate merge-fix first (see
appendix), and rebuild the 'pu' branch from scratch, starting at
the tip of 'jch'.
```

- Update "What's cooking" message to review the updates to existing topics, newly added topics and graduated topics.

```
This step is helped with Meta/cook script.

$ Meta/cook

This script inspects the history between master..pu, finds tips
of topic branches, compares what it found with the current
contents in Meta/whats-cooking.txt, and updates that file.
Topics not listed in the file but are found in master..pu are
added to the "New topics" section, topics listed in the file that
are no longer found in master..pu are moved to the "Graduated to
master" section, and topics whose commits changed their states
(e.g. used to be only in 'pu', now merged to 'next') are updated
with change markers "<<" and ">>".

Look for lines enclosed in "<<" and ">>"; they hold contents from
old file that are replaced by this integration round.  After
verifying them, remove the old part.  Review the description for
each topic and update its doneness and plan as needed.  To review
the updated plan, run

$ Meta/cook -w

which will pick up comments given to the topics, such as "Will
merge to 'next'", etc. (see Meta/cook script to learn what kind
of phrases are supported).
```

- Compile, test and install all four (five) integration branches; Meta/Dothem script may aid this step.
- Format documentation if the *master* branch was updated; Meta/dodoc.sh script may aid this step.
- Push the integration branches out to public places; Meta/pushall script may aid this step.

Observations

Some observations to be made.

- Each topic is tested individually, and also together with other topics cooking first in *pu*, then in *jch* and then in *next*. Until it matures, no part of it is merged to *master*.
- A topic already in *next* can get fixes while still in *next*. Such a topic will have many merges to *next* (in other words, "git log --first-parent next" will show many "Merge branch *ai/topic* to next" for the same topic.
- An unobvious fix for *maint* is cooked in *next* and then merged to *master* to make extra sure it is Ok and then merged to *maint*.
- Even when *next* becomes empty (in other words, all topics prove stable and are merged to *master* and "git diff master next" shows empty), it has tons of merge commits that will never be in *master*.
- In principle, "git log --first-parent master..next" should show nothing but merges (in practice, there are fixup commits and reverts that are not merges).
- Commits near the tip of a topic branch that are not in *next* are fair game to be discarded, replaced or rewritten. Commits already merged to *next* will not be.
- Being in the *next* branch is not a guarantee for a topic to be included in the next feature release. Being in the *master* branch typically is.

Appendix

Preparing a "merge-fix"

A merge of two topics may not textually conflict but still have conflict at the semantic level. A classic example is for one topic to rename an variable and all its uses, while another topic adds a new use of the variable under its old name. When these two topics are merged together, the reference to the variable newly added by the latter topic will still use the old name in the result.

The Meta/Reintegrate script that is used by redo-jch and redo-pu scripts implements a crude but usable way to work this issue around. When the script merges branch $X, it checks if "refs/merge-fix/$X" exists, and if so, the effect of it is squashed into the result of the mechanical merge. In other words,

```
$ echo $X | Meta/Reintegrate
```

is roughly equivalent to this sequence:

```
$ git merge --rerere-autoupdate $X
$ git commit
$ git cherry-pick -n refs/merge-fix/$X
$ git commit --amend
```

The goal of this "prepare a merge-fix" step is to come up with a commit that can be squashed into a result of mechanical merge to correct semantic conflicts.

After finding that the result of merging branch "ai/topic" to an integration branch had such a semantic conflict, say pu~4, check the problematic merge out on a detached HEAD, edit the working tree to fix the semantic conflict, and make a separate commit to record the fix-up:

```
$ git checkout pu~4
$ git show -s --pretty=%s ;# double check
Merge branch 'ai/topic' to pu
$ edit
$ git commit -m 'merge-fix/ai/topic' -a
```

Then make a reference "refs/merge-fix/ai/topic" to point at this result:

```
$ git update-ref refs/merge-fix/ai/topic HEAD
```

Then double check the result by asking Meta/Reintegrate to redo the merge:

```
$ git checkout pu~5 ;# the parent of the problem merge
$ echo ai/topic | Meta/Reintegrate
$ git diff pu~4
```

This time, because you prepared refs/merge-fix/ai/topic, the resulting merge should have been tweaked to include the fix for the semantic conflict.

Note that this assumes that the order in which conflicting branches are merged does not change. If the reason why merging ai/topic branch needs this merge-fix is because another branch merged earlier to the integration branch changed the underlying assumption ai/topic branch made (e.g. ai/topic branch added a site to refer to a variable, while the other branch renamed that variable and adjusted existing use sites), and if you changed redo-jch (or redo-pu) script to merge ai/topic branch before the other branch, then the above merge-fix should not be applied while merging ai/topic, but should instead be applied while merging the other branch. You would need to move the fix to apply to the other branch, perhaps like this:

```
$ mf=refs/merge-fix
$ git update-ref $mf/$the_other_branch $mf/ai/topic
$ git update-ref -d $mf/ai/topic
```

Last updated 2015-05-03 21:26:09 CEST

How to integrate new subcommands

This is how-to documentation for people who want to add extension commands to Git. It should be read alongside api-builtin.txt.

Runtime environment

Git subcommands are standalone executables that live in the Git exec path, normally /usr/lib/git-core. The git executable itself is a thin wrapper that knows where the subcommands live, and runs them by passing command-line arguments to them.

(If "git foo" is not found in the Git exec path, the wrapper will look in the rest of your $PATH for it. Thus, it's possible to write local Git extensions that don't live in system space.)

Implementation languages

Most subcommands are written in C or shell. A few are written in Perl.

While we strongly encourage coding in portable C for portability, these specific scripting languages are also acceptable. We won't accept more without a very strong technical case, as we don't want to broaden the Git suite's required dependencies. Import utilities, surgical tools, remote helpers and other code at the edges of the Git suite are more lenient and we allow Python (and even Tcl/tk), but they should not be used for core functions.

This may change in the future. Especially Python is not allowed in core because we need better Python integration in the Git Windows installer before we can be confident people in that environment won't experience an unacceptably large loss of capability.

C commands are normally written as single modules, named after the command, that link a collection of functions called libgit. Thus, your command *git-foo* would normally be implemented as a single "git-foo.c" (or "builtin/foo.c" if it is to be linked to the main binary); this organization makes it easy for people reading the code to find things.

See the CodingGuidelines document for other guidance on what we consider good practice in C and shell, and api-builtin.txt for the support functions available to built-in commands written in C.

What every extension command needs

You must have a man page, written in asciidoc (this is what Git help followed by your subcommand name will display). Be aware that there is a local asciidoc configuration and macros which you should use. It's often helpful to start by cloning an existing page and replacing the text content.

You must have a test, written to report in TAP (Test Anything Protocol). Tests are executables (usually shell scripts) that live in the *t* subdirectory of the tree. Each test name begins with *t* and a sequence number that controls where in the test sequence it will be executed; conventionally the rest of the name stem is that of the command being tested.

Read the file t/README to learn more about the conventions to be used in writing tests, and the test support library.

Integrating a command

Here are the things you need to do when you want to merge a new subcommand into the Git tree.

1. Don't forget to sign off your patch!
2. Append your command name to one of the variables BUILTIN_OBJS, EXTRA_PROGRAMS, SCRIPT_SH, SCRIPT_PERL or SCRIPT_PYTHON.
3. Drop its test in the t directory.
4. If your command is implemented in an interpreted language with a p-code intermediate form, make sure .gitignore in the main directory includes a pattern entry that ignores such files. Python .pyc and .pyo files will already be covered.
5. If your command has any dependency on a particular version of your language, document it in the INSTALL file.
6. There is a file command-list.txt in the distribution main directory that categorizes commands by type, so they can be listed in appropriate subsections in the documentation's summary command list. Add an entry for yours. To understand the categories, look at git-commands.txt in the main directory.
7. Give the maintainer one paragraph to include in the RelNotes file to describe the new feature; a good place to do so is in the cover letter [PATCH 0/n].

That's all there is to it.

Last updated 2015-05-03 21:25:54 CEST

How to rebase from an internal branch

```
Petr Baudis <pasky@suse.cz> writes:

> Dear diary, on Sun, Aug 14, 2005 at 09:57:13AM CEST, I got a letter
> where Junio C Hamano <junkio@cox.net> told me that...
>> Linus Torvalds <torvalds@osdl.org> writes:
>>
>> > Junio, maybe you want to talk about how you move patches from your "pu"
>> > branch to the real branches.
>>
> Actually, wouldn't this be also precisely for what StGIT is intended to?
```

Exactly my feeling. I was sort of waiting for Catalin to speak up. With its basing philosophical ancestry on quilt, this is the kind of task StGIT is designed to do.

I just have done a simpler one, this time using only the core Git tools.

I had a handful of commits that were ahead of master in pu, and I wanted to add some documentation bypassing my usual habit of placing new things in pu first. At the beginning, the commit ancestry graph looked like this:

```
                             *"pu" head
master --> #1 --> #2 --> #3
```

So I started from master, made a bunch of edits, and committed:

```
$ git checkout master
$ cd Documentation; ed git.txt ...
$ cd ..; git add Documentation/*.txt
$ git commit -s
```

After the commit, the ancestry graph would look like this:

```
                              *"pu" head
master^ --> #1 --> #2 --> #3
      \
        \---> master
```

The old master is now master^ (the first parent of the master). The new master commit holds my documentation updates.

Now I have to deal with "pu" branch.

This is the kind of situation I used to have all the time when Linus was the maintainer and I was a contributor, when you look at "master" branch being the "maintainer" branch, and "pu" branch being the "contributor" branch. Your work started at the tip of the "maintainer" branch some time ago, you made a lot of progress in the meantime, and now the maintainer branch has some other commits you do not have yet. And "git rebase" was written with the explicit purpose of helping to maintain branches like "pu". You *could* merge master to pu and keep going, but if you eventually want to cherrypick and merge some but not necessarily all changes back to the master branch, it often makes later operations for *you* easier if you rebase (i.e. carry forward your changes) "pu" rather than merge. So I ran "git rebase":

```
$ git checkout pu
$ git rebase master pu
```

What this does is to pick all the commits since the current branch (note that I now am on "pu" branch) forked from the master branch, and forward port these changes.

```
master^ --> #1 --> #2 --> #3
      \                                  *"pu" head
        \---> master --> #1' --> #2' --> #3'
```

The diff between master^ and #1 is applied to master and committed to create #1' commit with the commit information (log, author and date) taken from commit #1. On top of that #2' and #3' commits are made similarly out of #2 and #3 commits.

Old #3 is not recorded in any of the .git/refs/heads/ file anymore, so after doing this you will have dangling commit if you ran fsck-cache, which is normal. After testing "pu", you can run "git prune" to get rid of those original three commits.

While I am talking about "git rebase", I should talk about how to do cherrypicking using only the core Git tools.

Let’s go back to the earlier picture, with different labels.

You, as an individual developer, cloned upstream repository and made a couple of commits on top of it.

```
                               *your "master" head
upstream --> #1 --> #2 --> #3
```

You would want changes #2 and #3 incorporated in the upstream, while you feel that #1 may need further improvements. So you prepare #2 and #3 for e-mail submission.

```
$ git format-patch master^^ master
```

This creates two files, 0001-XXXX.patch and 0002-XXXX.patch. Send them out "To: " your project maintainer and "Cc: " your mailing list. You could use contributed script git-send-email if your host has necessary perl modules for this, but your usual MUA would do as long as it does not corrupt whitespaces in the patch.

Then you would wait, and you find out that the upstream picked up your changes, along with other changes.

```
 where                      *your "master" head
upstream --> #1 --> #2 --> #3
  used   \
 to be     \--> #A --> #2' --> #3' --> #B --> #C
                                              *upstream head
```

The two commits #2' and #3' in the above picture record the same changes your e-mail submission for #2 and #3 contained, but probably with the new sign-off line added by the upstream maintainer and definitely with different committer and ancestry information, they are different objects from #2 and #3 commits.

You fetch from upstream, but not merge.

```
$ git fetch upstream
```

This leaves the updated upstream head in .git/FETCH_HEAD but does not touch your .git/HEAD or .git/refs/heads/master. You run "git rebase" now.

```
$ git rebase FETCH_HEAD master
```

Earlier, I said that rebase applies all the commits from your branch on top of the upstream head. Well, I lied. "git rebase" is a bit smarter than that and notices that #2 and #3 need not be applied, so it only applies #1. The commit ancestry graph becomes something like this:

```
 where                     *your old "master" head
upstream --> #1 --> #2 --> #3
  used   \                     your new "master" head*
 to be     \--> #A --> #2' --> #3' --> #B --> #C --> #1'
                                              *upstream
                                              head
```

Again, "git prune" would discard the disused commits #1-#3 and you continue on starting from the new "master" head, which is the #1' commit.

-jc

Last updated 2015-05-03 21:26:08 CEST

How to rebuild from update hook

The pages under http://www.kernel.org/pub/software/scm/git/docs/ are built from Documentation/ directory of the git.git project and needed to be kept up-to-date. The www.kernel.org/ servers are mirrored and I was told that the origin of the mirror is on the machine $some.kernel.org, on which I was given an account when I took over Git maintainership from Linus.

The directories relevant to this how-to are these two:

```
/pub/scm/git/git.git/       The public Git repository.
/pub/software/scm/git/docs/ The HTML documentation page.
```

So I made a repository to generate the documentation under my home directory over there.

```
$ cd
$ mkdir doc-git && cd doc-git
$ git clone /pub/scm/git/git.git/ docgen
```

What needs to happen is to update the $HOME/doc-git/docgen/ working tree, build HTML docs there and install the result in /pub/software/scm/git/docs/ directory. So I wrote a little script:

```
$ cat >dododoc.sh <<\EOF
#!/bin/sh
cd $HOME/doc-git/docgen || exit
```

```
unset GIT_DIR

git pull /pub/scm/git/git.git/ master &&
cd Documentation &&
make install-webdoc
EOF
```

Initially I used to run this by hand whenever I push into the public Git repository. Then I did a cron job that ran twice a day. The current round uses the post-update hook mechanism, like this:

```
$ cat >/pub/scm/git/git.git/hooks/post-update <<\EOF
#!/bin/sh
#
# An example hook script to prepare a packed repository for use over
# dumb transports.
#
# To enable this hook, make this file executable by "chmod +x post-update".

case " $* " in
*' refs/heads/master '*)
        echo $HOME/doc-git/dododoc.sh | at now
        ;;
esac
exec git-update-server-info
EOF
$ chmod +x /pub/scm/git/git.git/hooks/post-update
```

There are four things worth mentioning:

- The update-hook is run after the repository accepts a "git push", under my user privilege. It is given the full names of refs that have been updated as arguments. My post-update runs the dododoc.sh script only when the master head is updated.
- When update-hook is run, GIT_DIR is set to . by the calling receive-pack. This is inherited by the dododoc.sh run via the "at" command, and needs to be unset; otherwise, "git pull" it does into $HOME/doc-git/docgen/ repository would not work correctly.
- The stdout of update hook script is not connected to git push; I run the heavy part of the command inside "at", to receive the execution report via e-mail.
- This is still crude and does not protect against simultaneous make invocations stomping on each other. I would need to add some locking mechanism for this.

Last updated 2015-05-03 21:26:07 CEST

How to recover a corrupted blob object

```
On Fri, 9 Nov 2007, Yossi Leybovich wrote:
>
> Did not help still the repository look for this object?
> Any one know how can I track this object and understand which file is it
```

So exactly **because** the SHA-1 hash is cryptographically secure, the hash itself doesn't actually tell you anything, in order to fix a corrupt object you basically have to find the "original source" for it.

The easiest way to do that is almost always to have backups, and find the same object somewhere else. Backups really are a good idea, and Git makes it pretty easy (if nothing else, just clone the repository somewhere else, and make sure that you do **not** use a hard-linked clone, and preferably not the same disk/machine).

But since you don't seem to have backups right now, the good news is that especially with a single blob being corrupt, these things **are** somewhat debuggable.

First off, move the corrupt object away, and **save** it. The most common cause of corruption so far has been memory corruption, but even so, there are people who would be interested in seeing the corruption - but it's basically impossible to judge the corruption until we can also see the original object, so right now the corrupt object is useless, but it's very interesting for the future, in the hope that you can re-create a non-corrupt version.

```
So:

> ib]$ mv .git/objects/4b/9458b3786228369c63936db65827de3cc06200 ../
```

This is the right thing to do, although it's usually best to save it under it's full SHA-1 name (you just dropped the "4b" from the result ;).

Let’s see what that tells us:

```
> ib]$ git-fsck --full
> broken link from    tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8
>              to    blob 4b9458b3786228369c63936db65827de3cc06200
> missing blob 4b9458b3786228369c63936db65827de3cc06200
```

Ok, I removed the "dangling commit" messages, because they are just messages about the fact that you probably have rebased etc, so they’re not at all interesting. But what remains is still very useful. In particular, we now know which tree points to it!

Now you can do

```
git ls-tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8
```

which will show something like

```
100644 blob 8d14531846b95bfa3564b58ccfb7913a034323b8    .gitignore
100644 blob ebf9bf84da0aab5ed944264a5db2a65fe3a3e883    .mailmap
100644 blob ca442d313d86dc67e0a2e5d584b465bd382cbf5c    COPYING
100644 blob ee909f2cc49e54f0799a4739d24c4cb9151ae453    CREDITS
040000 tree 0f5f709c17ad89e72bdbbef6ea221c69807009f6    Documentation
100644 blob 1570d248ad9237e4fa6e4d079336b9da62d9ba32    Kbuild
100644 blob 1c7c229a092665b11cd46a25dbd40feeb31661d9    MAINTAINERS
...
```

and you should now have a line that looks like

```
10064 blob 4b9458b3786228369c63936db65827de3cc06200     my-magic-file
```

in the output. This already tells you a **lot** it tells you what file the corrupt blob came from!

Now, it doesn’t tell you quite enough, though: it doesn’t tell what **version** of the file didn’t get correctly written! You might be really lucky, and it may be the version that you already have checked out in your working tree, in which case fixing this problem is really simple, just do

```
git hash-object -w my-magic-file
```

again, and if it outputs the missing SHA-1 (4b945..) you’re now all done!

But that’s the really lucky case, so let’s assume that it was some older version that was broken. How do you tell which version it was?

The easiest way to do it is to do

```
git log --raw --all --full-history -- subdirectory/my-magic-file
```

and that will show you the whole log for that file (please realize that the tree you had may not be the top-level tree, so you need to figure out which subdirectory it was in on your own), and because you’re asking for raw output, you’ll now get something like

```
commit abc
Author:
Date:
  ..
:100644 100644 4b9458b... newsha... M  somedirectory/my-magic-file


commit xyz
Author:
Date:


  ..
:100644 100644 oldsha... 4b9458b... M   somedirectory/my-magic-file
```

and this actually tells you what the **previous** and **subsequent** versions of that file were! So now you can look at those ("oldsha" and "newsha" respectively), and hopefully you have done commits often, and can re-create the missing my-magic-file version by looking at those older and newer versions!

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.

Trying to recreate trees and especially commits is **much** harder. So you were lucky that it’s a blob. It’s quite possible that you can recreate the thing.

```
Linus
```

Last updated 2015-05-03 21:26:05 CEST

How to recover an object from scratch

I was recently presented with a repository with a corrupted packfile, and was asked if the data was recoverable. This post-mortem describes the steps I took to investigate and fix the problem. I thought others might find the process interesting, and it might help somebody in the same situation.

> Note: In this case, no good copy of the repository was available. For the much easier case where you can get the corrupted object from elsewhere, see this howto.

I started with an fsck, which found a problem with exactly one object (I’ve used $pack and $obj below to keep the output readable, and also because I’ll refer to them later):

```
$ git fsck
error: $pack SHA1 checksum mismatch
error: index CRC mismatch for object $obj from $pack at offset 51653873
error: inflate: data stream error (incorrect data check)
error: cannot unpack $obj from $pack at offset 51653873
```

The pack checksum failing means a byte is munged somewhere, and it is presumably in the object mentioned (since both the index checksum and zlib were failing).

Reading the zlib source code, I found that "incorrect data check" means that the adler-32 checksum at the end of the zlib data did not match the inflated data. So stepping the data through zlib would not help, as it did not fail until the very end, when we realize the CRC does not match. The problematic bytes could be anywhere in the object data.

The first thing I did was pull the broken data out of the packfile. I needed to know how big the object was, which I found out with:

```
$ git show-index <$idx | cut -d' ' -f1 | sort -n | grep -A1 51653873
51653873
51664736
```

Show-index gives us the list of objects and their offsets. We throw away everything but the offsets, and then sort them so that our interesting offset (which we got from the fsck output above) is followed immediately by the offset of the next object. Now we know that the object data is 10863 bytes long, and we can grab it with:

```
dd if=$pack of=object bs=1 skip=51653873 count=10863
```

I inspected a hexdump of the data, looking for any obvious bogosity (e.g., a 4K run of zeroes would be a good sign of filesystem corruption). But everything looked pretty reasonable.

Note that the "object" file isn’t fit for feeding straight to zlib; it has the git packed object header, which is variable-length. We want to strip that off so we can start playing with the zlib data directly. You can either work your way through it manually (the format is described in Documentation/technical/pack-format.txt), or you can walk through it in a debugger. I did the latter, creating a valid pack like:

```
# pack magic and version
printf 'PACK\0\0\0\2' >tmp.pack
# pack has one object
printf '\0\0\0\1' >>tmp.pack
# now add our object data
cat object >>tmp.pack
# and then append the pack trailer
/path/to/git.git/test-sha1 -b <tmp.pack >trailer
cat trailer >>tmp.pack
```

and then running "git index-pack tmp.pack" in the debugger (stop at unpack_raw_entry). Doing this, I found that there were 3 bytes of header (and the header itself had a sane type and size). So I stripped those off with:

```
    dd if=object of=zlib bs=1 skip=3
```

I ran the result through zlib's inflate using a custom C program. And while it did report the error, I did get the right number of output bytes (i.e., it matched git's size header that we decoded above). But feeding the result back to "git hash-object" didn't produce the same sha1. So there were some wrong bytes, but I didn't know which. The file happened to be C source code, so I hoped I could notice something obviously wrong with it, but I didn't. I even got it to compile!

I also tried comparing it to other versions of the same path in the repository, hoping that there would be some part of the diff that didn't make sense. Unfortunately, this happened to be the only revision of this particular file in the repository, so I had nothing to compare against.

So I took a different approach. Working under the guess that the corruption was limited to a single byte, I wrote a program to munge each byte individually, and try inflating the result. Since the object was only 10K compressed, that worked out to about 2.5M attempts, which took a few minutes.

The program I used is here:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <zlib.h>

static int try_zlib(unsigned char *buf, int len)
{
        /* make this absurdly large so we don't have to loop */
        static unsigned char out[1024*1024];
        z_stream z;
        int ret;

        memset(&z, 0, sizeof(z));
        inflateInit(&z);

        z.next_in = buf;
        z.avail_in = len;
        z.next_out = out;
        z.avail_out = sizeof(out);

        ret = inflate(&z, 0);
        inflateEnd(&z);
        return ret >= 0;
}

/* eye candy */
static int counter = 0;
static void progress(int sig)
{
        fprintf(stderr, "\r%d", counter);
        alarm(1);
}

int main(void)
{
        /* oversized so we can read the whole buffer in */
        unsigned char buf[1024*1024];
        int len;
        unsigned i, j;

        signal(SIGALRM, progress);
        alarm(1);

        len = read(0, buf, sizeof(buf));
        for (i = 0; i < len; i++) {
                unsigned char c = buf[i];
                for (j = 0; j <= 0xff; j++) {
                        buf[i] = j;

                        counter++;
                        if (try_zlib(buf, len))
                                printf("i=%d, j=%x\n", i, j);
                }
                buf[i] = c;
        }

        alarm(0);
        fprintf(stderr, "\n");
        return 0;
}
```

I compiled and ran with:

```
   gcc -Wall -Werror -O3 munge.c -o munge -lz
   ./munge <zlib
```

There were a few false positives early on (if you write "no data" in the zlib header, zlib thinks it's just fine :)). But I got a hit about halfway through:

```
i=5642, j=c7
```

I let it run to completion, and got a few more hits at the end (where it was munging the CRC to match our broken data). So there was a good chance this middle hit was the source of the problem.

I confirmed by tweaking the byte in a hex editor, zlib inflating the result (no errors!), and then piping the output into "git hash-object", which reported the sha1 of the broken object. Success!

I fixed the packfile itself with:

```
chmod +w $pack
printf '\xc7' | dd of=$pack bs=1 seek=51659518 conv=notrunc
chmod -w $pack
```

The `\xc7` comes from the replacement byte our "munge" program found. The offset 51659518 is derived by taking the original object offset (51653873), adding the replacement offset found by "munge" (5642), and then adding back in the 3 bytes of git header we stripped.

After that, "git fsck" ran clean.

As for the corruption itself, I was lucky that it was indeed a single byte. In fact, it turned out to be a single bit. The byte 0xc7 was corrupted to 0xc5. So presumably it was caused by faulty hardware, or a cosmic ray.

And the aborted attempt to look at the inflated output to see what was wrong? I could have looked forever and never found it. Here's the diff between what the corrupted data inflates to, versus the real data:

```
-       cp = strtok (arg, "+");
+       cp = strtok (arg, ".");
```

It tweaked one byte and still ended up as valid, readable C that just happened to do something totally different! One takeaway is that on a less unlucky day, looking at the zlib output might have actually been helpful, as most random changes would actually break the C code.

But more importantly, git's hashing and checksumming noticed a problem that easily could have gone undetected in another system. The result still compiled, but would have caused an interesting bug (that would have been blamed on some random commit).

The adventure continues…

I ended up doing this again! Same entity, new hardware. The assumption at this point is that the old disk corrupted the packfile, and then the corruption was migrated to the new hardware (because it was done by rsync or similar, and no fsck was done at the time of migration).

This time, the affected blob was over 20 megabytes, which was far too large to do a brute-force on. I followed the instructions above to create the `zlib` file. I then used the `inflate` program below to pull the corrupted data from that. Examining that output gave me a hint about where in the file the corruption was. But now I was working with the file itself, not the zlib contents. So knowing the sha1 of the object and the approximate area of the corruption, I used the `sha1-munge` program below to brute-force the correct byte.

Here's the inflate program (it's essentially `gunzip` but without the `.gz` header processing):

```
#include <stdio.h>
#include <string.h>
#include <zlib.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        /*
         * oversized so we can read the whole buffer in;
         * this could actually be switched to streaming
         * to avoid any memory limitations
         */
        static unsigned char buf[25 * 1024 * 1024];
        static unsigned char out[25 * 1024 * 1024];
        int len;
        z_stream z;
        int ret;

        len = read(0, buf, sizeof(buf));
        memset(&z, 0, sizeof(z));
        inflateInit(&z);

        z.next_in = buf;
        z.avail_in = len;
        z.next_out = out;
        z.avail_out = sizeof(out);
        ret = inflate(&z, 0);
        if (ret != Z_OK && ret != Z_STREAM_END)
                fprintf(stderr, "initial inflate failed (%d)\n", ret);

        fprintf(stderr, "outputting %lu bytes", z.total_out);
        fwrite(out, 1, z.total_out, stdout);
        return 0;
}
```

And here is the `sha1-munge` program:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <openssl/sha.h>
#include <stdlib.h>

/* eye candy */
static int counter = 0;
static void progress(int sig)
{
        fprintf(stderr, "\r%d", counter);
        alarm(1);
}

static const signed char hexval_table[256] = {
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 00-07 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 08-0f */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 10-17 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 18-1f */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 20-27 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 28-2f */
          0,  1,  2,  3,  4,  5,  6,  7,                /* 30-37 */
          8,  9, -1, -1, -1, -1, -1, -1,                /* 38-3f */
         -1, 10, 11, 12, 13, 14, 15, -1,                /* 40-47 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 48-4f */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 50-57 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 58-5f */
         -1, 10, 11, 12, 13, 14, 15, -1,                /* 60-67 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 68-67 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 70-77 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 78-7f */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 80-87 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 88-8f */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 90-97 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* 98-9f */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* a0-a7 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* a8-af */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* b0-b7 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* b8-bf */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* c0-c7 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* c8-cf */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* d0-d7 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* d8-df */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* e0-e7 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* e8-ef */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* f0-f7 */
         -1, -1, -1, -1, -1, -1, -1, -1,                /* f8-ff */
};

static inline unsigned int hexval(unsigned char c)
{
return hexval_table[c];
}

static int get_sha1_hex(const char *hex, unsigned char *sha1)
{
        int i;
        for (i = 0; i < 20; i++) {
                unsigned int val;
                /*
                 * hex[1]=='\0' is caught when val is checked below,
                 * but if hex[0] is NUL we have to avoid reading
                 * past the end of the string:
                 */
                if (!hex[0])
                        return -1;
                val = (hexval(hex[0]) << 4) | hexval(hex[1]);
                if (val & ~0xff)
                        return -1;
                *sha1++ = val;
                hex += 2;
        }
        return 0;
}

int main(int argc, char **argv)
{
        /* oversized so we can read the whole buffer in */
        static unsigned char buf[25 * 1024 * 1024];
        char header[32];
        int header_len;
        unsigned char have[20], want[20];
        int start, len;
        SHA_CTX orig;
        unsigned i, j;

        if (!argv[1] || get_sha1_hex(argv[1], want)) {
                fprintf(stderr, "usage: sha1-munge <sha1> [start] <file.in\n");
                return 1;
        }

        if (argv[2])
                start = atoi(argv[2]);
        else
                start = 0;

        len = read(0, buf, sizeof(buf));
        header_len = sprintf(header, "blob %d", len) + 1;
        fprintf(stderr, "using header: %s\n", header);

        /*
         * We keep a running sha1 so that if you are munging
         * near the end of the file, we do not have to re-sha1
         * the unchanged earlier bytes
         */
        SHA1_Init(&orig);
        SHA1_Update(&orig, header, header_len);
        if (start)
                SHA1_Update(&orig, buf, start);

        signal(SIGALRM, progress);
        alarm(1);

        for (i = start; i < len; i++) {
                unsigned char c;
                SHA_CTX x;

#if 0
                /*
                 * deletion -- this would not actually work in practice,
                 * I think, because we've already committed to a
                 * particular size in the header. Ditto for addition
                 * below. In those cases, you'd have to do the whole
                 * sha1 from scratch, or possibly keep three running
                 * "orig" sha1 computations going.
                 */
                memcpy(&x, &orig, sizeof(x));
                SHA1_Update(&x, buf + i + 1, len - i - 1);
                SHA1_Final(have, &x);
                if (!memcmp(have, want, 20))
                        printf("i=%d, deletion\n", i);
#endif

                /*
                 * replacement -- note that this tries each of the 256
                 * possible bytes. If you suspect a single-bit flip,
                 * it would be much shorter to just try the 8
                 * bit-flipped variants.
                 */
                c = buf[i];
                for (j = 0; j <= 0xff; j++) {
                        buf[i] = j;

                        memcpy(&x, &orig, sizeof(x));
                        SHA1_Update(&x, buf + i, len - i);
                        SHA1_Final(have, &x);
                        if (!memcmp(have, want, 20))
                                printf("i=%d, j=%02x\n", i, j);
                }
                buf[i] = c;

#if 0
                /* addition */
                for (j = 0; j <= 0xff; j++) {
                        unsigned char extra = j;
                        memcpy(&x, &orig, sizeof(x));
                        SHA1_Update(&x, &extra, 1);
                        SHA1_Update(&x, buf + i, len - i);
                        SHA1_Final(have, &x);
                        if (!memcmp(have, want, 20))
                                printf("i=%d, addition=%02x", i, j);
                }
#endif

                SHA1_Update(&orig, buf + i, 1);
                counter++;
        }

        alarm(0);
        fprintf(stderr, "\r%d\n", counter);
        return 0;
}
```

Last updated 2015-05-03 21:26:06 CEST

How to revert a faulty merge

Alan <alan@clueserver.org> said:

```
I have a master branch.  We have a branch off of that that some
developers are doing work on.  They claim it is ready. We merge it
into the master branch.  It breaks something so we revert the merge.
They make changes to the code.  they get it to a point where they say
it is ok and we merge again.

When examined, we find that code changes made before the revert are
not in the master branch, but code changes after are in the master
branch.
```

and asked for help recovering from this situation.

The history immediately after the "revert of the merge" would look like this:

```
---o---o---o---M---x---x---W
              /
      ---A---B
```

where A and B are on the side development that was not so good, M is the merge that brings these premature changes into the mainline, x are changes unrelated to what the side branch did and already made on the mainline, and W is the "revert of the merge M" (doesn't W look M upside down?). IOW, `"diff W^..W"` is similar to `"diff -R M^..M"`.

Such a "revert" of a merge can be made with:

```
$ git revert -m 1 M
```

After the developers of the side branch fix their mistakes, the history may look like this:

```
---o---o---o---M---x---x---W---x
              /
      ---A---B-------------------C---D
```

where C and D are to fix what was broken in A and B, and you may already have some other changes on the mainline after W.

If you merge the updated side branch (with D at its tip), none of the changes made in A or B will be in the result, because they were reverted by W. That is what Alan saw.

Linus explains the situation:

```
Reverting a regular commit just effectively undoes what that commit
did, and is fairly straightforward. But reverting a merge commit also
undoes the _data_ that the commit changed, but it does absolutely
nothing to the effects on _history_ that the merge had.

So the merge will still exist, and it will still be seen as joining
the two branches together, and future merges will see that merge as
the last shared state - and the revert that reverted the merge brought
in will not affect that at all.

So a "revert" undoes the data changes, but it's very much _not_ an
"undo" in the sense that it doesn't undo the effects of a commit on
the repository history.

So if you think of "revert" as "undo", then you're going to always
miss this part of reverts. Yes, it undoes the data, but no, it doesn't
undo history.
```

In such a situation, you would want to first revert the previous revert, which would make the history look like this:

```
---o---o---o---M---x---x---W---x---Y
              /
      ---A---B-------------------C---D
```

where Y is the revert of W. Such a "revert of the revert" can be done with:

```
$ git revert W
```

This history would (ignoring possible conflicts between what W and W..Y changed) be equivalent to not having W or Y at all in the history:

```
---o---o---o---M---x---x-------x----
              /
      ---A---B-------------------C---D
```

and merging the side branch again will not have conflict arising from an earlier revert and revert of the revert.

```
---o---o---o---M---x---x-------x-------*
              /                       /
      ---A---B-------------------C---D
```

Of course the changes made in C and D still can conflict with what was done by any of the x, but that is just a normal merge conflict.

On the other hand, if the developers of the side branch discarded their faulty A and B, and redone the changes on top of the updated mainline after the revert, the history would have looked like this:

```
---o---o---o---M---x---x---W---x---x
              /                 \
      ---A---B                   A'--B'--C'
```

If you reverted the revert in such a case as in the previous example:

```
---o---o---o---M---x---x---W---x---x---Y---*
              /                 \         /
      ---A---B                   A'--B'--C'
```

where Y is the revert of W, A' and B' are rerolled A and B, and there may also be a further fix-up C' on the side branch. `"diff Y^..Y"` is similar to `"diff -R W^..W"` (which in turn means it is similar to `"diff M^..M"`), and `"diff A'^..C'"` by definition would be similar but different from that, because it is a rerolled series of the earlier change. There will be a lot of overlapping changes that result in conflicts. So do not do "revert of revert" blindly without thinking..

```
---o---o---o---M---x---x---W---x---x
              /                 \
      ---A---B                   A'--B'--C'
```

In the history with rebased side branch, W (and M) are behind the merge base of the updated branch and the tip of the mainline, and they should merge without the past faulty merge and its revert getting in the way.

To recap, these are two very different scenarios, and they want two very different resolution strategies:

- If the faulty side branch was fixed by adding corrections on top, then doing a revert of the previous revert would be the right thing to do.
- If the faulty side branch whose effects were discarded by an earlier revert of a merge was rebuilt from scratch (i.e. rebasing and fixing, as you seem to have interpreted), then re-merging the result without doing anything else fancy would be the right thing to do. (See the ADDENDUM below for how to rebuild a branch from scratch without changing its original branching-off point.)

However, there are things to keep in mind when reverting a merge (and reverting such a revert).

For example, think about what reverting a merge (and then reverting the revert) does to bisectability. Ignore the fact that the revert of a revert is undoing it - just think of it as a "single commit that does a lot". Because that is what it does.

When you have a problem you are chasing down, and you hit a "revert this merge", what you're hitting is essentially a single commit that contains all the changes (but obviously in reverse) of all the commits that got merged. So it's debugging hell, because now you don't have lots of small changes that you can try to pinpoint which *part* of it changes.

But does it all work? Sure it does. You can revert a merge, and from a purely technical angle, Git did it very naturally and had no real troubles. It just considered it a change from "state before merge" to "state after merge", and that was it. Nothing complicated, nothing odd, nothing really dangerous. Git will do it without even thinking about it.

So from a technical angle, there's nothing wrong with reverting a merge, but from a workflow angle it's something that you generally should try to avoid.

If at all possible, for example, if you find a problem that got merged into the main tree, rather than revert the merge, try *really* hard to bisect the problem down into the branch you merged, and just fix it, or try to revert the individual commit that caused it.

Yes, it's more complex, and no, it's not always going to work (sometimes the answer is: "oops, I really shouldn't have merged it, because it wasn't ready yet, and I really need to undo *all* of the merge"). So then you really should revert the merge, but when you want to re-do the merge, you now need to do it by reverting the revert.

ADDENDUM

Sometimes you have to rewrite one of a topic branch's commits **and** you can't change the topic's branching-off point. Consider the following situation:

```
P---o---o---M---x---x---W---x
```

```
 \         /
  A---B---C
```

where commit W reverted commit M because it turned out that commit B was wrong and needs to be rewritten, but you need the rewritten topic to still branch from commit P (perhaps P is a branching-off point for yet another branch, and you want be able to merge the topic into both branches).

The natural thing to do in this case is to checkout the A-B-C branch and use "rebase -i P" to change commit B. However this does not rewrite commit A, because "rebase -i" by default fast-forwards over any initial commits selected with the "pick" command. So you end up with this:

```
P---o---o---M---x---x---W---x
 \         /
  A---B---C   <-- old branch
   \
    B'---C'   <-- naively rewritten branch
```

To merge A-B'-C' into the mainline branch you would still have to first revert commit W in order to pick up the changes in A, but then it’s likely that the changes in B' will conflict with the original B changes re-introduced by the reversion of W.

However, you can avoid these problems if you recreate the entire branch, including commit A:

```
  A'---B'---C'  <-- completely rewritten branch
 /
P---o---o---M---x---x---W---x
 \         /
  A---B---C
```

You can merge A'-B'-C' into the mainline branch without worrying about first reverting W. Mainline’s history would look like this:

```
  A'---B'---C'------------------
 /                               \
P---o---o---M---x---x---W---x---M2
 \         /
  A---B---C
```

But if you don’t actually need to change commit A, then you need some way to recreate it as a new commit with the same changes in it. The rebase command’s --no-ff option provides a way to do this:

```
$ git rebase [-i] --no-ff P
```

The --no-ff option creates a new branch A'-B'-C' with all-new commits (all the SHA IDs will be different) even if in the interactive case you only actually modify commit B. You can then merge this new branch directly into the mainline branch and be sure you’ll get all of the branch’s changes.

You can also use --no-ff in cases where you just add extra commits to the topic to fix it up. Let’s revisit the situation discussed at the start of this howto:

```
P---o---o---M---x---x---W---x
 \         /
  A---B---C----------------D---E   <-- fixed-up topic branch
```

At this point, you can use --no-ff to recreate the topic branch:

```
$ git checkout E
$ git rebase --no-ff P
```

yielding

```
  A'---B'---C'------------D'---E'  <-- recreated topic branch
 /
P---o---o---M---x---x---W---x
 \         /
  A---B---C----------------D---E
```

You can merge the recreated branch into the mainline without reverting commit W, and mainline’s history will look like this:

```
  A'---B'---C'------------D'---E'
 /                               \
P---o---o---M---x---x---W---x---M2
 \         /
  A---B---C
```

Last updated 2015-05-03 21:26:04 CEST

How to revert an existing commit

One of the changes I pulled into the *master* branch turns out to break building Git with GCC 2.95. While they were well-intentioned portability fixes, keeping things working with gcc-2.95 was also important. Here is what I did to revert the change in the *master* branch and to adjust the *pu* branch, using core Git tools and barebone Porcelain.

First, prepare a throw-away branch in case I screw things up.

```
$ git checkout -b revert-c99 master
```

Now I am on the *revert-c99* branch. Let's figure out which commit to revert. I happen to know that the top of the *master* branch is a merge, and its second parent (i.e. foreign commit I merged from) has the change I would want to undo. Further I happen to know that that merge introduced 5 commits or so:

```
$ git show-branch --more=4 master master^2 | head
* [master] Merge refs/heads/portable from http://www.cs.berkeley....
 ! [master^2] Replace C99 array initializers with code.
--
-  [master] Merge refs/heads/portable from http://www.cs.berkeley....
*+ [master^2] Replace C99 array initializers with code.
*+ [master^2~1] Replace unsetenv() and setenv() with older putenv().
*+ [master^2~2] Include sys/time.h in daemon.c.
*+ [master^2~3] Fix ?: statements.
*+ [master^2~4] Replace zero-length array decls with [].
*  [master~1] tutorial note about git branch
```

The *--more=4* above means "after we reach the merge base of refs, show until we display four more common commits". That last commit would have been where the "portable" branch was forked from the main git.git repository, so this would show everything on both branches since then. I just limited the output to the first handful using *head*.

Now I know *master^2~4* (pronounce it as "find the second parent of the *master*, and then go four generations back following the first parent") is the one I would want to revert. Since I also want to say why I am reverting it, the *-n* flag is given to *git revert*. This prevents it from actually making a commit, and instead *git revert* leaves the commit log message it wanted to use in *.msg* file:

```
$ git revert -n master^2~4
$ cat .msg
Revert "Replace zero-length array decls with []."

This reverts 6c5f9baa3bc0d63e141e0afc23110205379905a4 commit.
$ git diff HEAD ;# to make sure what we are reverting makes sense.
$ make CC=gcc-2.95 clean test ;# make sure it fixed the breakage.
$ make clean test ;# make sure it did not cause other breakage.
```

The reverted change makes sense (from reading the *diff* output), does fix the problem (from *make CC=gcc-2.95* test), and does not cause new breakage (from the last *make test*). I'm ready to commit:

```
$ git commit -a -s ;# read .msg into the log,
                    # and explain why I am reverting.
```

I could have screwed up in any of the above steps, but in the worst case I could just have done *git checkout master* to start over. Fortunately I did not have to; what I have in the current branch *revert-c99* is what I want. So merge that back into *master*:

```
$ git checkout master
$ git merge revert-c99 ;# this should be a fast-forward
Updating from 10d781b9caa4f71495c7b34963bef137216f86a8 to e3a693c...
 cache.h        |    8 ++++----
 commit.c       |    2 +-
 ls-files.c     |    2 +-
 receive-pack.c |    2 +-
 server-info.c  |    2 +-
 5 files changed, 8 insertions(+), 8 deletions(-)
```

There is no need to redo the test at this point. We fast-forwarded and we know *master* matches *revert-c99* exactly. In fact:

```
$ git diff master..revert-c99
```

says nothing.

Then we rebase the *pu* branch as usual.

```
$ git checkout pu
$ git tag pu-anchor pu
$ git rebase master
* Applying: Redo "revert" using three-way merge machinery.
First trying simple merge strategy to cherry-pick.
* Applying: Remove git-apply-patch-script.
First trying simple merge strategy to cherry-pick.
Simple cherry-pick fails; trying Automatic cherry-pick.
Removing Documentation/git-apply-patch-script.txt
Removing git-apply-patch-script
* Applying: Document "git cherry-pick" and "git revert"
First trying simple merge strategy to cherry-pick.
* Applying: mailinfo and applymbox updates
First trying simple merge strategy to cherry-pick.
* Applying: Show commits in topo order and name all commits.
First trying simple merge strategy to cherry-pick.
* Applying: More documentation updates.
First trying simple merge strategy to cherry-pick.
```

The temporary tag *pu-anchor* is me just being careful, in case *git rebase* screws up. After this, I can do these for sanity check:

```
$ git diff pu-anchor..pu ;# make sure we got the master fix.
$ make CC=gcc-2.95 clean test ;# make sure it fixed the breakage.
$ make clean test ;# make sure it did not cause other breakage.
```

Everything is in the good order. I do not need the temporary branch or tag anymore, so remove them:

```
$ rm -f .git/refs/tags/pu-anchor
$ git branch -d revert-c99
```

It was an emergency fix, so we might as well merge it into the *release candidate* branch, although I expect the next release would be some days off:

```
$ git checkout rc
$ git pull . master
Packing 0 objects
Unpacking 0 objects

* commit-ish: e3a693c...        refs/heads/master from .
Trying to merge e3a693c... into 8c1f5f0... using 10d781b...
Committed merge 7fb9b7262a1d1e0a47bbfdcbbcf50ce0635d3f8f
 cache.h        |    8 ++++----
 commit.c       |    2 +-
 ls-files.c     |    2 +-
 receive-pack.c |    2 +-
 server-info.c  |    2 +-
 5 files changed, 8 insertions(+), 8 deletions(-)
```

And the final repository status looks like this:

```
$ git show-branch --more=1 master pu rc
! [master] Revert "Replace zero-length array decls with []."
 ! [pu] git-repack: Add option to repack all objects.
  * [rc] Merge refs/heads/master from .
---
 +  [pu] git-repack: Add option to repack all objects.
 +  [pu~1] More documentation updates.
 +  [pu~2] Show commits in topo order and name all commits.
 +  [pu~3] mailinfo and applymbox updates
 +  [pu~4] Document "git cherry-pick" and "git revert"
 +  [pu~5] Remove git-apply-patch-script.
 +  [pu~6] Redo "revert" using three-way merge machinery.
  - [rc] Merge refs/heads/master from .
++* [master] Revert "Replace zero-length array decls with []."
  - [rc~1] Merge refs/heads/master from .
... [master~1] Merge refs/heads/portable from http://www.cs.berkeley....
```

Last updated 2015-05-03 21:25:56 CEST

How to separate topic branches

This text was originally a footnote to a discussion about the behaviour of the git diff commands.

Often I find myself doing that [running diff against something other than HEAD] while rewriting messy development history. For example, I start doing some work without knowing exactly where it leads, and end up with a history like this:

```
    "master"
o---o
     \                    "topic"
      o---o---o---o---o---o
```

At this point, "topic" contains something I know I want, but it contains two concepts that turned out to be completely independent. And often, one topic component is larger than the other. It may contain more than two topics.

In order to rewrite this mess to be more manageable, I would first do "diff master..topic", to extract the changes into a single patch, start picking pieces from it to get logically self-contained units, and start building on top of "master":

```
$ git diff master..topic >P.diff
$ git checkout -b topicA master
... pick and apply pieces from P.diff to build
... commits on topicA branch.


      o---o---o
     /        "topicA"
o---o"master"
     \                    "topic"
      o---o---o---o---o---o
```

Before doing each commit on "topicA" HEAD, I run "diff HEAD" before update-index the affected paths, or "diff --cached HEAD" after. Also I would run "diff --cached master" to make sure that the changes are only the ones related to "topicA". Usually I do this for smaller topics first.

After that, I'd do the remainder of the original "topic", but for that, I do not start from the patchfile I extracted by comparing "master" and "topic" I used initially. Still on "topicA", I extract "diff topic", and use it to rebuild the other topic:

```
$ git diff -R topic >P.diff ;# --cached also would work fine
$ git checkout -b topicB master
... pick and apply pieces from P.diff to build
... commits on topicB branch.


                        "topicB"
       o---o---o---o---o
      /
     /o---o---o
    |/        "topicA"
o---o"master"
     \                    "topic"
      o---o---o---o---o---o
```

After I am done, I'd try a pretend-merge between "topicA" and "topicB" in order to make sure I have not missed anything:

```
$ git pull . topicA ;# merge it into current "topicB"
$ git diff topic
                        "topicB"
       o---o---o---o---o---* (pretend merge)
      /                   /
     /o---o---o----------'
    |/        "topicA"
o---o"master"
     \                    "topic"
      o---o---o---o---o---o
```

The last diff better not to show anything other than cleanups for crufts. Then I can finally clean things up:

```
$ git branch -D topic
$ git reset --hard HEAD^ ;# nuke pretend merge


                        "topicB"
       o---o---o---o---o
      /
     /o---o---o
    |/        "topicA"
o---o"master"
```

Last updated 2015-05-03 21:26:04 CEST

How to setup Git server over http

Note This document is from 2006. A lot has happened since then, and this document is now relevant mainly if your web host is not CGI capable. Almost everyone else should instead look at git-http-backend(1).

Since Apache is one of those packages people like to compile themselves while others prefer the bureaucrat's dream Debian, it is impossible to give guidelines which will work for everyone. Just send some feedback to the mailing list at git@vger.kernel.org to get this document tailored to your favorite distro.

What's needed:

- Have an Apache web-server

  ```
  On Debian:
    $ apt-get install apache2
    To get apache2 by default started,
    edit /etc/default/apache2 and set NO_START=0
  ```

- can edit the configuration of it.

  ```
  This could be found under /etc/httpd, or refer to your Apache documentation.

  On Debian: this means being able to edit files under /etc/apache2
  ```

- can restart it.

  ```
  'apachectl --graceful' might do. If it doesn't, just stop and
  restart apache. Be warning that active connections to your server
  might be aborted by this.

  On Debian:
    $ /etc/init.d/apache2 restart
  or
    $ /etc/init.d/apache2 force-reload
    (which seems to do the same)
  This adds symlinks from the /etc/apache2/mods-enabled to
  /etc/apache2/mods-available.
  ```

- have permissions to chown a directory
- have Git installed on the client, and
- either have Git installed on the server or have a webdav client on the client.

In effect, this means you're going to be root, or that you're using a preconfigured WebDAV server.

Step 1: setup a bare Git repository

At the time of writing, git-http-push cannot remotely create a Git repository. So we have to do that at the server side with Git. Another option is to generate an empty bare repository at the client and copy it to the server with a WebDAV client (which is the only option if Git is not installed on the server).

Create the directory under the DocumentRoot of the directories served by Apache. As an example we take /usr/local/apache2, but try "grep DocumentRoot /where/ever/httpd.conf" to find your root:

```
$ cd /usr/local/apache/htdocs
$ mkdir my-new-repo.git
```

```
On Debian:

$ cd /var/www
$ mkdir my-new-repo.git
```

Initialize a bare repository

```
$ cd my-new-repo.git
$ git --bare init
```

Change the ownership to your web-server's credentials. Use `"grep ^User httpd.conf"` and `"grep ^Group httpd.conf"` to find out:

```
$ chown -R www.www .
```

```
On Debian:

$ chown -R www-data.www-data .
```

If you do not know which user Apache runs as, you can alternatively do a "chmod -R a+w .", inspect the files which are created later on, and set the permissions appropriately.

Restart apache2, and check whether http://server/my-new-repo.git gives a directory listing. If not, check whether apache started up successfully.

Step 2: enable DAV on this repository

First make sure the dav_module is loaded. For this, insert in httpd.conf:

```
LoadModule dav_module libexec/httpd/libdav.so
AddModule mod_dav.c
```

Also make sure that this line exists which is the file used for locking DAV operations:

```
DAVLockDB "/usr/local/apache2/temp/DAV.lock"

On Debian these steps can be performed with:

Enable the dav and dav_fs modules of apache:
$ a2enmod dav_fs
(just to be sure. dav_fs might be unneeded, I don't know)
$ a2enmod dav
The DAV lock is located in /etc/apache2/mods-available/dav_fs.conf:
  DAVLockDB /var/lock/apache2/DAVLock
```

Of course, it can point somewhere else, but the string is actually just a prefix in some Apache configurations, and therefore the *directory* has to be writable by the user Apache runs as.

Then, add something like this to your httpd.conf

```
<Location /my-new-repo.git>
   DAV on
   AuthType Basic
   AuthName "Git"
   AuthUserFile /usr/local/apache2/conf/passwd.git
   Require valid-user
</Location>

On Debian:
  Create (or add to) /etc/apache2/conf.d/git.conf :

<Location /my-new-repo.git>
   DAV on
   AuthType Basic
   AuthName "Git"
   AuthUserFile /etc/apache2/passwd.git
   Require valid-user
</Location>

Debian automatically reads all files under /etc/apache2/conf.d.
```

The password file can be somewhere else, but it has to be readable by Apache and preferably not readable by the world.

Create this file by $ htpasswd -c /usr/local/apache2/conf/passwd.git <user>

```
On Debian:
  $ htpasswd -c /etc/apache2/passwd.git <user>
```

You will be asked a password, and the file is created. Subsequent calls to htpasswd should omit the *-c* option, since you want to append to the existing file.

You need to restart Apache.

Now go to http://<username>@<servername>/my-new-repo.git in your browser to check whether it asks for a password and accepts the right password.

On Debian:

```
To test the WebDAV part, do:

$ apt-get install litmus
$ litmus http://<servername>/my-new-repo.git <username> <password>

Most tests should pass.
```

A command-line tool to test WebDAV is cadaver. If you prefer GUIs, for example, konqueror can open WebDAV URLs as "webdav://..." or "webdavs://...".

If you're into Windows, from XP onwards Internet Explorer supports WebDAV. For this, do Internet Explorer → Open Location → http://<servername>/my-new-repo.git [x] Open as webfolder → login .

Step 3: setup the client

Make sure that you have HTTP support, i.e. your Git was built with libcurl (version more recent than 7.10). The command *git http-push* with no argument should display a usage message.

Then, add the following to your $HOME/.netrc (you can do without, but will be asked to input your password a *lot* of times):

```
machine <servername>
login <username>
password <password>
```

...and set permissions: chmod 600 ~/.netrc

If you want to access the web-server by its IP, you have to type that in, instead of the server name.

To check whether all is OK, do:

```
curl --netrc --location -v http://<username>@<servername>/my-new-repo.git/HEAD
```

...this should give something like *ref: refs/heads/master*, which is the content of the file HEAD on the server.

Now, add the remote in your existing repository which contains the project you want to export:

```
$ git-config remote.upload.url \
    http://<username>@<servername>/my-new-repo.git/
```

It is important to put the last */*; Without it, the server will send a redirect which git-http-push does not (yet) understand, and git-http-push will repeat the request infinitely.

Step 4: make the initial push

From your client repository, do

```
$ git push upload master
```

This pushes branch *master* (which is assumed to be the branch you want to export) to repository called *upload*, which we previously defined with git-config.

Using a proxy:

If you have to access the WebDAV server from behind an HTTP(S) proxy, set the variable *all_proxy* to *http://proxy-host.com:port*, or *http://login-on-proxy:passwd-on-proxy@proxy-host.com:port*. See *man curl* for details.

Troubleshooting:

If git-http-push says

```
Error: no DAV locking support on remote repo http://...
```

then it means the web-server did not accept your authentication. Make sure that the user name and password matches in httpd.conf, .netrc and the URL you are uploading to.

If git-http-push shows you an error (22/502) when trying to MOVE a blob, it means that your web-server somehow does not recognize its name in the request; This can happen when you start Apache, but then disable the network interface. A simple restart of Apache helps.

Errors like (22/502) are of format (curl error code/http error code). So (22/404) means something like *not found* at the server.

Reading /usr/local/apache2/logs/error_log is often helpful.

```
On Debian: Read /var/log/apache2/error.log instead.
```

If you access HTTPS locations, Git may fail verifying the SSL certificate (this is return code 60). Setting http.sslVerify=false can help diagnosing the problem, but removes security checks.

Debian References: http://www.debian-administration.org/articles/285

Authors Johannes Schindelin <Johannes.Schindelin@gmx.de> Rutger Nijlunsing <git@wingding.demon.nl> Matthieu Moy <Matthieu.Moy@imag.fr>

Last updated 2015-05-03 21:26:02 CEST

How to use the update hook

When your developer runs git-push into the repository, git-receive-pack is run (either locally or over ssh) as that developer, so is hooks/update script. Quoting from the relevant section of the documentation:

```
Before each ref is updated, if $GIT_DIR/hooks/update file exists
and executable, it is called 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".  Two sha1 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.
```

So if your policy is (1) always require fast-forward push (i.e. never allow "git-push repo +branch:branch"), (2) you have a list of users allowed to update each branch, and (3) you do not let tags to be overwritten, then you can use something like this as your hooks/update script.

[jc: editorial note. This is a much improved version by Carl since I posted the original outline]

```
#!/bin/bash

umask 002

# If you are having trouble with this access control hook script
# you can try setting this to true.  It will tell you exactly
# why a user is being allowed/denied access.

verbose=false

# Default shell globbing messes things up downstream
GLOBIGNORE=*

function grant {
  $verbose && echo >&2 "-Grant-         $1"
  echo grant
  exit 0
}

function deny {
  $verbose && echo >&2 "-Deny-          $1"
  echo deny
  exit 1
}

function info {
  $verbose && echo >&2 "-Info-          $1"
}

# Implement generic branch and tag policies.
# - Tags should not be updated once created.
# - Branches should only be fast-forwarded unless their pattern starts with '+'
case "$1" in
  refs/tags/*)
    git rev-parse --verify -q "$1" &&
    deny >/dev/null "You can't overwrite an existing tag"
    ;;
  refs/heads/*)
    # No rebasing or rewinding
    if expr "$2" : '0*$' >/dev/null; then
      info "The branch '$1' is new..."
    else
      # updating -- make sure it is a fast-forward
      mb=$(git-merge-base "$2" "$3")
      case "$mb,$2" in
        "$2,$mb") info "Update is fast-forward" ;;
        *)        noff=y; info "This is not a fast-forward update.";;
      esac
    fi
    ;;
  *)
    deny >/dev/null \
    "Branch is not under refs/heads or refs/tags.  What are you trying to do?"
    ;;
esac

# Implement per-branch controls based on username
allowed_users_file=$GIT_DIR/info/allowed-users
username=$(id -u -n)
info "The user is: '$username'"

if test -f "$allowed_users_file"
then
  rc=$(cat $allowed_users_file | grep -v '^#' | grep -v '^$' |
    while read heads user_patterns
    do
      # does this rule apply to us?
      head_pattern=${heads#+}
      matchlen=$(expr "$1" : "${head_pattern#+}")
      test "$matchlen" = ${#1} || continue

      # if non-ff, $heads must be with the '+' prefix
      test -n "$noff" &&
      test "$head_pattern" = "$heads" && continue

      info "Found matching head pattern: '$head_pattern'"
      for user_pattern in $user_patterns; do
        info "Checking user: '$username' against pattern: '$user_pattern'"
        matchlen=$(expr "$username" : "$user_pattern")
        if test "$matchlen" = "${#username}"
        then
          grant "Allowing user: '$username' with pattern: '$user_pattern'"
        fi
      done
      deny "The user is not in the access list for this branch"
    done
  )
  case "$rc" in
    grant) grant >/dev/null "Granting access based on $allowed_users_file" ;;
    deny)  deny  >/dev/null "Denying  access based on $allowed_users_file" ;;
    *) ;;
  esac
fi

allowed_groups_file=$GIT_DIR/info/allowed-groups
groups=$(id -G -n)
info "The user belongs to the following groups:"
info "'$groups'"

if test -f "$allowed_groups_file"
then
  rc=$(cat $allowed_groups_file | grep -v '^#' | grep -v '^$' |
    while read heads group_patterns
    do
      # does this rule apply to us?
      head_pattern=${heads#+}
      matchlen=$(expr "$1" : "${head_pattern#+}")
      test "$matchlen" = ${#1} || continue

      # if non-ff, $heads must be with the '+' prefix
      test -n "$noff" &&
      test "$head_pattern" = "$heads" && continue

      info "Found matching head pattern: '$head_pattern'"
      for group_pattern in $group_patterns; do
        for groupname in $groups; do
          info "Checking group: '$groupname' against pattern: '$group_pattern'"
          matchlen=$(expr "$groupname" : "$group_pattern")
          if test "$matchlen" = "${#groupname}"
          then
            grant "Allowing group: '$groupname' with pattern: '$group_pattern'"
          fi
        done
      done
      deny "None of the user's groups are in the access list for this branch"
    done
  )
  case "$rc" in
    grant) grant >/dev/null "Granting access based on $allowed_groups_file" ;;
    deny)  deny  >/dev/null "Denying  access based on $allowed_groups_file" ;;
    *) ;;
  esac
fi

deny >/dev/null "There are no more rules to check.  Denying access"
```

This uses two files, $GIT_DIR/info/allowed-users and allowed-groups, to describe which heads can be pushed into by whom. The format of each file would look like this:

```
refs/heads/master   junio
+refs/heads/pu      junio
```

```
refs/heads/cogito$  pasky
refs/heads/bw/.*    linus
refs/heads/tmp/.*   .*
refs/tags/v[0-9].*  junio
```

With this, Linus can push or create "bw/penguin" or "bw/zebra" or "bw/panda" branches, Pasky can do only "cogito", and JC can do master and pu branches and make versioned tags. And anybody can do tmp/blah branches. The `+` sign at the pu record means that JC can make non-fast-forward pushes on it.

Last updated 2015-05-03 21:26:01 CEST

How to use git-daemon

Git can be run in inetd mode and in stand alone mode. But all you want is let a coworker pull from you, and therefore need to set up a Git server real quick, right?

Note that git-daemon is not really chatty at the moment, especially when things do not go according to plan (e.g. a socket could not be bound).

Another word of warning: if you run

```
$ git ls-remote git://127.0.0.1/rule-the-world.git
```

and you see a message like

```
fatal: The remote end hung up unexpectedly
```

it only means that *something* went wrong. To find out *what* went wrong, you have to ask the server. (Git refuses to be more precise for your security only. Take off your shoes now. You have any coins in your pockets? Sorry, not allowed — who knows what you planned to do with them?)

With these two caveats, let's see an example:

```
$ git daemon --reuseaddr --verbose --base-path=/home/gitte/git \
  --export-all -- /home/gitte/git/rule-the-world.git
```

(Of course, unless your user name is `gitte` *and* your repository is in ~/rule-the-world.git, you have to adjust the paths. If your repository is not bare, be aware that you have to type the path to the .git directory!)

This invocation tries to reuse the address if it is already taken (this can save you some debugging, because otherwise killing and restarting git-daemon could just silently fail to bind to a socket).

Also, it is (relatively) verbose when somebody actually connects to it. It also sets the base path, which means that all the projects which can be accessed using this daemon have to reside in or under that path.

The option `--export-all` just means that you *don't* have to create a file named `git-daemon-export-ok` in each exported repository. (Otherwise, git-daemon would complain loudly, and refuse to cooperate.)

Last of all, the repository which should be exported is specified. It is a good practice to put the paths after a "--" separator.

Now, test your daemon with

```
$ git ls-remote git://127.0.0.1/rule-the-world.git
```

If this does not work, find out why, and submit a patch to this document.

Last updated 2015-05-03 21:26:00 CEST

How to use the subtree merge strategy

There are situations where you want to include contents in your project from an independently developed project. You can just pull from the other project as long as there are no conflicting paths.

The problematic case is when there are conflicting files. Potential candidates are Makefiles and other standard filenames. You could merge these files but probably you do not want to. A better solution for this problem can be to merge the project as its own subdirectory. This is not supported by the *recursive* merge strategy, so just pulling won't work.

What you want is the *subtree* merge strategy, which helps you in such a situation.

In this example, let's say you have the repository at `/path/to/B` (but it can be a URL as well, if you want). You want to merge the *master* branch of that repository to the `dir-B` subdirectory in your current branch.

Here is the command sequence you need:

```
$ git remote add -f Bproject /path/to/B <1>
$ git merge -s ours --no-commit Bproject/master <2>
$ git read-tree --prefix=dir-B/ -u Bproject/master <3>
$ git commit -m "Merge B project as our subdirectory" <4>

$ git pull -s subtree Bproject master <5>
```

1. name the other project "Bproject", and fetch.
2. prepare for the later step to record the result as a merge.
3. read "master" branch of Bproject to the subdirectory "dir-B".
4. record the merge result.
5. maintain the result with subsequent merges using "subtree"

The first four commands are used for the initial merge, while the last one is to merge updates from *B project*.

Comparing *subtree* merge with submodules

- The benefit of using subtree merge is that it requires less administrative burden from the users of your repository. It works with older (before Git v1.5.2) clients and you have the code right after clone.
- However if you use submodules then you can choose not to transfer the submodule objects. This may be a problem with the subtree merge.
- Also, in case you make changes to the other project, it is easier to submit changes if you just use submodules.

Additional tips

- If you made changes to the other project in your repository, they may want to merge from your project. This is possible using subtree — it can shift up the paths in your tree and then they can merge only the relevant parts of your tree.
- Please note that if the other project merges from you, then it will connect its history to yours, which can be something they don't want to.

Last updated 2015-05-03 21:25:57 CEST

How to use a signed tag in pull requests

A typical distributed workflow using Git is for a contributor to fork a project, build on it, publish the result to her public repository, and ask the "upstream" person (often the owner of the project where she forked from) to pull from her public repository. Requesting such a "pull" is made easy by the `git request-pull` command.

Earlier, a typical pull request may have started like this:

```
 The following changes since commit 406da78032179...:

   Froboz 3.2 (2011-09-30 14:20:57 -0700)

 are available in the Git repository at:

   example.com:/git/froboz.git for-xyzzy
```

followed by a shortlog of the changes and a diffstat.

The request was for a branch name (e.g. `for-xyzzy`) in the public repository of the contributor, and even though it stated where the contributor forked her work from, the message did not say anything about the commit to expect at the tip of the for-xyzzy branch. If the site that hosts the public repository of the contributor cannot be fully trusted, it was unnecessarily hard to make sure what was pulled by the integrator was genuinely what the contributor had produced for the project. Also there was no easy way for third-party auditors to later verify the resulting history.

Starting from Git release v1.7.9, a contributor can add a signed tag to the commit at the tip of the history and ask the integrator to pull that signed tag. When the integrator runs `git pull`, the signed tag is automatically verified to assure that the history is not tampered with. In addition, the resulting merge commit records the content of the signed tag, so that other people can verify that the branch merged by the integrator was signed by the contributor, without fetching the signed tag used to validate the pull request separately and keeping it in the refs namespace.

This document describes the workflow between the contributor and the integrator, using Git v1.7.9 or later.

A contributor or a lieutenant

After preparing her work to be pulled, the contributor uses `git tag -s` to create a signed tag:

```
$ git checkout work
$ ... "git pull" from sublieutenants, "git commit" your own work ...
$ git tag -s -m "Completed frotz feature" frotz-for-xyzzy work
```

Note that this example uses the `-m` option to create a signed tag with just a one-liner message, but this is for illustration purposes only. It is advisable to compose a well-written explanation of what the topic does to justify why it is worthwhile for the integrator to pull it, as this message will eventually become part of the final history after the integrator responds to the pull request (as we will see later).

Then she pushes the tag out to her public repository:

```
$ git push example.com:/git/froboz.git/ +frotz-for-xyzzy
```

There is no need to push the `work` branch or anything else.

Note that the above command line used a plus sign at the beginning of `+frotz-for-xyzzy` to allow forcing the update of a tag, as the same contributor may want to reuse a signed tag with the same name after the previous pull request has already been responded to.

The contributor then prepares a message to request a "pull":

```
$ git request-pull v3.2 example.com:/git/froboz.git/ frotz-for-xyzzy >msg.txt
```

The arguments are:

1. the version of the integrator's commit the contributor based her work on;
2. the URL of the repository, to which the contributor has pushed what she wants to get pulled; and
3. the name of the tag the contributor wants to get pulled (earlier, she could write only a branch name here).

The resulting msg.txt file begins like so:

```
The following changes since commit 406da78032179...:

  Froboz 3.2 (2011-09-30 14:20:57 -0700)

are available in the Git repository at:

  example.com:/git/froboz.git tags/frotz-for-xyzzy

for you to fetch changes up to 703f05ad5835c...:

  Add tests and documentation for frotz (2011-12-02 10:02:52 -0800)

-----------------------------------------------
Completed frotz feature
-----------------------------------------------
```

followed by a shortlog of the changes and a diffstat. Comparing this with the earlier illustration of the output from the traditional `git request-pull` command, the reader should notice that:

1. The tip commit to expect is shown to the integrator; and
2. The signed tag message is shown prominently between the dashed lines before the shortlog.

The latter is why the contributor would want to justify why pulling her work is worthwhile when creating the signed tag. The contributor then opens her favorite MUA, reads msg.txt, edits and sends it to her upstream integrator.

Integrator

After receiving such a pull request message, the integrator fetches and integrates the tag named in the request, with:

```
$ git pull example.com:/git/froboz.git/ tags/frotz-for-xyzzy
```

This operation will always open an editor to allow the integrator to fine tune the commit log message when merging a signed tag. Also, pulling a signed tag will always create a merge commit even when the integrator does not have any new commit since the contributor's work forked (i.e. *fast forward*), so that the integrator can properly explain what the merge is about and why it was made.

In the editor, the integrator will see something like this:

```
Merge tag 'frotz-for-xyzzy' of example.com:/git/froboz.git/

Completed frotz feature
# gpg: Signature made Fri 02 Dec 2011 10:03:01 AM PST using RSA key ID 96AFE6CB
# gpg: Good signature from "Con Tributor <nitfol@example.com>"
```

Notice that the message recorded in the signed tag "Completed frotz feature" appears here, and again that is why it is important for the contributor to explain her work well when creating the signed tag.

As usual, the lines commented with `#` are stripped out. The resulting commit records the signed tag used for this validation in a hidden field so that it can later be used by others to audit the history. There is no need for the integrator to keep a separate copy of the tag in his repository (i.e. `git tag -l` won't list the `frotz-for-xyzzy` tag in the above example), and there is no need to publish the tag to his public repository, either.

After the integrator responds to the pull request and her work becomes part of the permanent history, the contributor can remove the tag from her public repository, if she chooses, in order to keep the tag namespace of her public repository clean, with:

```
$ git push example.com:/git/froboz.git :frotz-for-xyzzy
```

Auditors

The `--show-signature` option can be given to `git log` or `git show` and shows the verification status of the embedded signed tag in merge commits created when the integrator responded to a pull request of a signed tag.

A typical output from `git show --show-signature` may look like this:

```
$ git show --show-signature
commit 02306ef6a3498a39118aef9df7975bdb50091585
merged tag 'frotz-for-xyzzy'
gpg: Signature made Fri 06 Jan 2012 12:41:49 PM PST using RSA key ID 96AFE6CB
gpg: Good signature from "Con Tributor <nitfol@example.com>"
Merge: 406da78 703f05a
Author: Inte Grator <xyzzy@example.com>
Date:   Tue Jan 17 13:49:41 2012 -0800

    Merge tag 'frotz-for-xyzzy' of example.com:/git/froboz.git/

    Completed frotz feature

    * tag 'frotz-for-xyzzy' (100 commits)
      Add tests and documentation for frotz
      ...
```

There is no need for the auditor to explicitly fetch the contributor's signature, or to even be aware of what tag(s) the contributor and integrator used to communicate the signature. All the required information is recorded as part of the merge commit.

Last updated 2015-05-03 21:25:58 CEST

Git API Documents

Git has grown a set of internal API over time. This collection documents them.

Last updated 2015-05-03 21:26:42 CEST

allocation growing API

Dynamically growing an array using realloc() is error prone and boring.

Define your array with:

- a pointer (`item`) that points at the array, initialized to `NULL` (although please name the variable based on its contents, not on its type);
- an integer variable (`alloc`) that keeps track of how big the current allocation is, initialized to `0`;
- another integer variable (`nr`) to keep track of how many elements the array currently has, initialized to `0`.

Then before adding `` n`th element to the item, call `ALLOC_GROW(item, n, alloc) ``. This ensures that the array can hold at least `n` elements by calling `realloc(3)` and adjusting `alloc` variable.

```
sometype *item;
size_t nr;
size_t alloc

for (i = 0; i < nr; i++)
        if (we like item[i] already)
                return;

/* we did not like any existing one, so add one */
ALLOC_GROW(item, nr + 1, alloc);
item[nr++] = value you like;
```

You are responsible for updating the `nr` variable.

If you need to specify the number of elements to allocate explicitly then use the macro `REALLOC_ARRAY(item, alloc)` instead of `ALLOC_GROW`.

Last updated 2014-11-27 19:58:08 CET

argv-array API

The argv-array API allows one to dynamically build and store NULL-terminated lists. An argv-array maintains the invariant that the `argv` member always points to a non-NULL array, and that the array is always NULL-terminated at the element pointed to by `argv[argc]`. This makes the result suitable for passing to functions expecting to receive argv from main(), or the run-command API.

The string-list API is similar, but cannot be used for these purposes; instead of storing a straight string pointer, it contains an item structure with a `util` field that is not compatible with the traditional argv interface.

Each `argv_array` manages its own memory. Any strings pushed into the array are duplicated, and all memory is freed by argv_array_clear().

Data Structures

`struct argv_array`
: A single array. This should be initialized by assignment from `ARGV_ARRAY_INIT`, or by calling `argv_array_init`. The `argv` member contains the actual array; the `argc` member contains the number of elements in the array, not including the terminating NULL.

Functions

`argv_array_init`
: Initialize an array. This is no different than assigning from `ARGV_ARRAY_INIT`.

`argv_array_push`
: Push a copy of a string onto the end of the array.

`argv_array_pushl`
: Push a list of strings onto the end of the array. The arguments should be a list of `const char *` strings, terminated by a NULL argument.

`argv_array_pushf`
: Format a string and push it onto the end of the array. This is a convenience wrapper combining `strbuf_addf` and `argv_array_push`.

`argv_array_pop`
: Remove the final element from the array. If there are no elements in the array, do nothing.

`argv_array_clear`
: Free all memory associated with the array and return it to the initial, empty state.

Last updated 2014-11-27 19:58:08 CET

builtin API

Adding a new built-in

There are 4 things to do to add a built-in command implementation to Git:

1. Define the implementation of the built-in command `foo` with signature:

   ```
   int cmd_foo(int argc, const char **argv, const char *prefix);
   ```

2. Add the external declaration for the function to `builtin.h`.
3. Add the command to the `commands[]` table defined in `git.c`. The entry should look like:

   ```
   { "foo", cmd_foo, <options> },
   ```

   where options is the bitwise-or of:

   `RUN_SETUP`
   : If there is not a Git directory to work on, abort. If there is a work tree, chdir to the top of it if the command was invoked in a subdirectory. If there is no work tree, no chdir() is done.

   `RUN_SETUP_GENTLY`
   : If there is a Git directory, chdir as per RUN_SETUP, otherwise, don't chdir anywhere.

   `USE_PAGER`
   : If the standard output is connected to a tty, spawn a pager and feed our output to it.

   `NEED_WORK_TREE`
   : Make sure there is a work tree, i.e. the command cannot act on bare repositories. This only makes sense when `RUN_SETUP` is also set.

4. Add `builtin/foo.o` to `BUILTIN_OBJS` in `Makefile`.

Additionally, if `foo` is a new command, there are 3 more things to do:

1. Add tests to `t/` directory.
2. Write documentation in `Documentation/git-foo.txt`.
3. Add an entry for `git-foo` to `command-list.txt`.
4. Add an entry for `/git-foo` to `.gitignore`.

How a built-in is called

The implementation `cmd_foo()` takes three parameters, `argc`, `argv, and `prefix`. The first two are similar to what `main()` of a standalone command would be called with.

When `RUN_SETUP` is specified in the `commands[]` table, and when you were started from a subdirectory of the work tree, `cmd_foo()` is called after chdir(2) to the top of the work tree, and `prefix` gets the path to the subdirectory the command started from. This allows you to convert a user-supplied pathname (typically relative to that directory) to a pathname relative to the top of the work tree.

The return value from `cmd_foo()` becomes the exit status of the command.

Last updated 2014-11-27 19:58:08 CET

config API

The config API gives callers a way to access Git configuration files (and files which have the same syntax). See git-config(1) for a discussion of the config file syntax.

General Usage

Config files are parsed linearly, and each variable found is passed to a caller-provided callback function. The callback function is responsible for any actions to be taken on the config option, and is free to ignore some options. It is not uncommon for the configuration to be parsed several times during the run of a Git program, with different callbacks picking out different variables useful to themselves.

A config callback function takes three parameters:

- the name of the parsed variable. This is in canonical "flat" form: the section, subsection, and variable segments will be separated by dots, and the section and variable segments will be all lowercase. E.g., `core.ignorecase`, `diff.SomeType.textconv`.
- the value of the found variable, as a string. If the variable had no value specified, the value will be NULL (typically this means it should be interpreted as boolean true).
- a void pointer passed in by the caller of the config API; this can contain callback-specific data

A config callback should return 0 for success, or -1 if the variable could not be parsed properly.

Basic Config Querying

Most programs will simply want to look up variables in all config files that Git knows about, using the normal precedence rules. To do this, call `git_config` with a callback function and void data pointer.

`git_config` will read all config sources in order of increasing priority. Thus a callback should typically overwrite previously-seen entries with new ones (e.g., if both the user-wide `~/.gitconfig` and repo-specific `.git/config` contain `color.ui`, the config machinery will first feed the user-wide one to the callback, and then the repo-specific one; by overwriting, the higher-priority repo-specific value is left at the end).

The `git_config_with_options` function lets the caller examine config while adjusting some of the default behavior of `git_config`. It should almost never be used by "regular" Git code that is looking up configuration variables. It is intended for advanced callers like `git-config`, which are intentionally tweaking the normal config-lookup process. It takes two extra parameters:

`filename`
: If this parameter is non-NULL, it specifies the name of a file to parse for configuration, rather than looking in the usual files. Regular `git_config` defaults to `NULL`.

`respect_includes`
: Specify whether include directives should be followed in parsed files. Regular `git_config` defaults to `1`.

There is a special version of `git_config` called `git_config_early`. This version takes an additional parameter to specify the repository config, instead of having it looked up via `git_path`. This is useful early in a Git program before the repository has been found. Unless you're working with early setup code, you probably don't want to use this.

Reading Specific Files

To read a specific file in git-config format, use `git_config_from_file`. This takes the same callback and data parameters as `git_config`.

Querying For Specific Variables

For programs wanting to query for specific variables in a non-callback manner, the config API provides two functions `git_config_get_value` and `git_config_get_value_multi`. They both read values from an internal cache generated previously from reading the config files.

`int git_config_get_value(const char *key, const char **value)`
: Finds the highest-priority value for the configuration variable `key`, stores the pointer to it in `value` and returns 0. When the configuration variable `key` is not found, returns 1 without touching `value`. The caller should not free or modify `value`, as it is owned by the cache.

`const struct string_list *git_config_get_value_multi(const char *key)`
: Finds and returns the value list, sorted in order of increasing priority for the configuration variable `key`. When the configuration variable `key` is not found, returns NULL. The caller should not free or modify the returned pointer, as it is owned by the cache.

`void git_config_clear(void)`
: Resets and invalidates the config cache.

The config API also provides type specific API functions which do conversion as well as retrieval for the queried variable, including:

`int git_config_get_int(const char *key, int *dest)`
: Finds and parses the value to an integer for the configuration variable `key`. Dies on error; otherwise, stores the value of the parsed integer in `dest` and returns 0. When the configuration variable `key` is not found, returns 1

without touching `dest`.

`int git_config_get_ulong(const char *key, unsigned long *dest)`
: Similar to `git_config_get_int` but for unsigned longs.

`int git_config_get_bool(const char *key, int *dest)`
: Finds and parses the value into a boolean value, for the configuration variable `key` respecting keywords like "true" and "false". Integer values are converted into true/false values (when they are non-zero or zero, respectively). Other values cause a die(). If parsing is successful, stores the value of the parsed result in `dest` and returns 0. When the configuration variable `key` is not found, returns 1 without touching `dest`.

`int git_config_get_bool_or_int(const char *key, int *is_bool, int *dest)`
: Similar to `git_config_get_bool`, except that integers are copied as-is, and `is_bool` flag is unset.

`int git_config_get_maybe_bool(const char *key, int *dest)`
: Similar to `git_config_get_bool`, except that it returns -1 on error rather than dying.

`int git_config_get_string_const(const char *key, const char **dest)`
: Allocates and copies the retrieved string into the `dest` parameter for the configuration variable `key`; if NULL string is given, prints an error message and returns -1. When the configuration variable `key` is not found, returns 1 without touching `dest`.

`int git_config_get_string(const char *key, char **dest)`
: Similar to `git_config_get_string_const`, except that retrieved value copied into the `dest` parameter is a mutable string.

`int git_config_get_pathname(const char *key, const char **dest)`
: Similar to `git_config_get_string`, but expands `~` or `~user` into the user's home directory when found at the beginning of the path.

`git_die_config(const char *key, const char *err, ...)`
: First prints the error message specified by the caller in `err` and then dies printing the line number and the file name of the highest priority value for the configuration variable `key`.

`void git_die_config_linenr(const char *key, const char *filename, int linenr)`
: Helper function which formats the die error message according to the parameters entered. Used by `git_die_config()`. It can be used by callers handling `git_config_get_value_multi()` to print the correct error message for the desired value.

See test-config.c for usage examples.

Value Parsing Helpers

To aid in parsing string values, the config API provides callbacks with a number of helper functions, including:

`git_config_int`
: Parse the string to an integer, including unit factors. Dies on error; otherwise, returns the parsed result.

`git_config_ulong`
: Identical to `git_config_int`, but for unsigned longs.

`git_config_bool`
: Parse a string into a boolean value, respecting keywords like "true" and "false". Integer values are converted into true/false values (when they are non-zero or zero, respectively). Other values cause a die(). If parsing is successful, the return value is the result.

`git_config_bool_or_int`
: Same as `git_config_bool`, except that integers are returned as-is, and an `is_bool` flag is unset.

`git_config_maybe_bool`
: Same as `git_config_bool`, except that it returns -1 on error rather than dying.

`git_config_string`
: Allocates and copies the value string into the `dest` parameter; if no string is given, prints an error message and returns -1.

`git_config_pathname`
: Similar to `git_config_string`, but expands `~` or `~user` into the user's home directory when found at the beginning of the path.

Include Directives

By default, the config parser does not respect include directives. However, a caller can use the special `git_config_include` wrapper callback to support them. To do so, you simply wrap your "real" callback function and data pointer in a `struct config_include_data`, and pass the wrapper to the regular config-reading functions. For example:

```
int read_file_with_include(const char *file, config_fn_t fn, void *data)
```

```
{
        struct config_include_data inc = CONFIG_INCLUDE_INIT;
        inc.fn = fn;
        inc.data = data;
        return git_config_from_file(git_config_include, file, &inc);
}
```

`git_config` respects includes automatically. The lower-level `git_config_from_file` does not.

Custom Configsets

A `config_set` can be used to construct an in-memory cache for config-like files that the caller specifies (i.e., files like `.gitmodules`, `~/.gitconfig` etc.). For example,

```
struct config_set gm_config;
git_configset_init(&gm_config);
int b;
/* we add config files to the config_set */
git_configset_add_file(&gm_config, ".gitmodules");
git_configset_add_file(&gm_config, ".gitmodules_alt");

if (!git_configset_get_bool(gm_config, "submodule.frotz.ignore", &b)) {
        /* hack hack hack */
}

/* when we are done with the configset */
git_configset_clear(&gm_config);
```

Configset API provides functions for the above mentioned work flow, including:

`void git_configset_init(struct config_set *cs)`
: Initializes the config_set `cs`.

`int git_configset_add_file(struct config_set *cs, const char *filename)`
: Parses the file and adds the variable-value pairs to the `config_set`, dies if there is an error in parsing the file. Returns 0 on success, or -1 if the file does not exist or is inaccessible. The user has to decide if he wants to free the incomplete configset or continue using it when the function returns -1.

`int git_configset_get_value(struct config_set *cs, const char *key, const char **value)`
: Finds the highest-priority value for the configuration variable `key` and config set `cs`, stores the pointer to it in `value` and returns 0. When the configuration variable `key` is not found, returns 1 without touching `value`. The caller should not free or modify `value`, as it is owned by the cache.

`const struct string_list *git_configset_get_value_multi(struct config_set *cs, const char *key)`
: Finds and returns the value list, sorted in order of increasing priority for the configuration variable `key` and config set `cs`. When the configuration variable `key` is not found, returns NULL. The caller should not free or modify the returned pointer, as it is owned by the cache.

`void git_configset_clear(struct config_set *cs)`
: Clears `config_set` structure, removes all saved variable-value pairs.

In addition to above functions, the `config_set` API provides type specific functions in the vein of `git_config_get_int` and family but with an extra parameter, pointer to struct `config_set`. They all behave similarly to the `git_config_get*()` family described in "Querying For Specific Variables" above.

Writing Config Files

Git gives multiple entry points in the Config API to write config values to files namely `git_config_set_in_file` and `git_config_set`, which write to a specific config file or to `.git/config` respectively. They both take a key/value pair as parameter. In the end they both call `git_config_set_multivar_in_file` which takes four parameters:

- the name of the file, as a string, to which key/value pairs will be written.
- the name of key, as a string. This is in canonical "flat" form: the section, subsection, and variable segments will be separated by dots, and the section and variable segments will be all lowercase. E.g., `core.ignorecase`, `diff.SomeType.textconv`.
- the value of the variable, as a string. If value is equal to NULL, it will remove the matching key from the config file.
- the value regex, as a string. It will disregard key/value pairs where value does not match.
- a multi_replace value, as an int. If value is equal to zero, nothing or only one matching key/value is replaced, else all matching key/values (regardless how many) are removed, before the new pair is written.

It returns 0 on success.

Also, there are functions `git_config_rename_section` and `git_config_rename_section_in_file` with parameters `old_name` and `new_name` for renaming or removing sections in the config files. If NULL is passed through `new_name` parameter, the section will be removed from the config file.

Last updated 2014-11-27 19:58:08 CET

credentials API

The credentials API provides an abstracted way of gathering username and password credentials from the user (even though credentials in the wider world can take many forms, in this document the word "credential" always refers to a username and password pair).

This document describes two interfaces: the C API that the credential subsystem provides to the rest of Git, and the protocol that Git uses to communicate with system-specific "credential helpers". If you are writing Git code that wants to look up or prompt for credentials, see the section "C API" below. If you want to write your own helper, see the section on "Credential Helpers" below.

Typical setup

```
+-----------------------+
| Git code (C)          |--- to server requiring --->
|                       |        authentication
|.......................|
| C credential API      |--- prompt ---> User
+-----------------------+
        ^      |
        | pipe |
        |      v
+-----------------------+
| Git credential helper |
+-----------------------+
```

The Git code (typically a remote-helper) will call the C API to obtain credential data like a login/password pair (credential_fill). The API will itself call a remote helper (e.g. "git credential-cache" or "git credential-store") that may retrieve credential data from a store. If the credential helper cannot find the information, the C API will prompt the user. Then, the caller of the API takes care of contacting the server, and does the actual authentication.

C API

The credential C API is meant to be called by Git code which needs to acquire or store a credential. It is centered around an object representing a single credential and provides three basic operations: fill (acquire credentials by calling helpers and/or prompting the user), approve (mark a credential as successfully used so that it can be stored for later use), and reject (mark a credential as unsuccessful so that it can be erased from any persistent storage).

Data Structures

`struct credential`

: This struct represents a single username/password combination along with any associated context. All string fields should be heap-allocated (or NULL if they are not known or not applicable). The meaning of the individual context fields is the same as their counterparts in the helper protocol; see the section below for a description of each field.

  The `helpers` member of the struct is a `string_list` of helpers. Each string specifies an external helper which will be run, in order, to either acquire or store credentials. See the section on credential helpers below. This list is filled-in by the API functions according to the corresponding configuration variables before consulting helpers, so there usually is no need for a caller to modify the helpers field at all.

  This struct should always be initialized with `CREDENTIAL_INIT` or `credential_init`.

Functions

`credential_init`

: Initialize a credential structure, setting all fields to empty.

`credential_clear`

Free any resources associated with the credential structure, returning it to a pristine initialized state.

`credential_fill`
: Instruct the credential subsystem to fill the username and password fields of the passed credential struct by first consulting helpers, then asking the user. After this function returns, the username and password fields of the credential are guaranteed to be non-NULL. If an error occurs, the function will die().

`credential_reject`
: Inform the credential subsystem that the provided credentials have been rejected. This will cause the credential subsystem to notify any helpers of the rejection (which allows them, for example, to purge the invalid credentials from storage). It will also free() the username and password fields of the credential and set them to NULL (readying the credential for another call to `credential_fill`). Any errors from helpers are ignored.

`credential_approve`
: Inform the credential subsystem that the provided credentials were successfully used for authentication. This will cause the credential subsystem to notify any helpers of the approval, so that they may store the result to be used again. Any errors from helpers are ignored.

`credential_from_url`
: Parse a URL into broken-down credential fields.

Example

The example below shows how the functions of the credential API could be used to login to a fictitious "foo" service on a remote host:

```
int foo_login(struct foo_connection *f)
{
        int status;
        /*
         * Create a credential with some context; we don't yet know the
         * username or password.
         */

        struct credential c = CREDENTIAL_INIT;
        c.protocol = xstrdup("foo");
        c.host = xstrdup(f->hostname);

        /*
         * Fill in the username and password fields by contacting
         * helpers and/or asking the user. The function will die if it
         * fails.
         */
        credential_fill(&c);

        /*
         * Otherwise, we have a username and password. Try to use it.
         */
        status = send_foo_login(f, c.username, c.password);
        switch (status) {
        case FOO_OK:
                /* It worked. Store the credential for later use. */
                credential_accept(&c);
                break;
        case FOO_BAD_LOGIN:
                /* Erase the credential from storage so we don't try it
                 * again. */
                credential_reject(&c);
                break;
        default:
                /*
                 * Some other error occurred. We don't know if the
                 * credential is good or bad, so report nothing to the
                 * credential subsystem.
                 */
        }

        /* Free any associated resources. */
        credential_clear(&c);

        return status;
}
```

Credential Helpers

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 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"
foo

# same as above, but pass an argument to the helper
foo --bar=baz

# the arguments are parsed by the shell, so use shell
# quoting if necessary
foo --bar="whitespace arg"

# you can also use an absolute path, which will not use the git wrapper
/path/to/my/helper --with-arguments

# or you can specify your own shell snippet
!f() { 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 a matching credential, 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 git-credential(7) for a detailed specification).

For a `get` operation, the helper should produce a list of attributes on stdout in the same format. 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. 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).

For a `store` or `erase` operation, the helper’s output is ignored. If it fails to perform the requested operation, it may complain to stderr to inform the user. If it does not support the requested operation (e.g., a read-only store), 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).

See also

gitcredentials(7)

git-config(5) (See configuration variables `credential.*`)

Last updated 2014-12-23 17:41:32 CET

decorate API

Talk about <decorate.h>

(Linus)

Last updated 2014-01-25 09:03:55 CET

diff API

The diff API is for programs that compare two sets of files (e.g. two trees, one tree and the index) and present the found difference in various ways. The calling program is responsible for feeding the API pairs of files, one from the "old" set and the corresponding one from "new" set, that are different. The library called through this API is called diffcore, and is responsible for two things.

- finding total rewrites (`-B`), renames (`-M`) and copies (`-C`), and changes that touch a string (`-S`), as specified by the caller.
- outputting the differences in various formats, as specified by the caller.

Calling sequence

- Prepare `struct diff_options` to record the set of diff options, and then call `diff_setup()` to initialize this structure. This sets up the vanilla default.
- Fill in the options structure to specify desired output format, rename detection, etc. `diff_opt_parse()` can be used to parse options given from the command line in a way consistent with existing git-diff family of programs.
- Call `diff_setup_done()`; this inspects the options set up so far for internal consistency and make necessary tweaking to it (e.g. if textual patch output was asked, recursive behaviour is turned on); the callback set_default in diff_options can be used to tweak this more.
- As you find different pairs of files, call `diff_change()` to feed modified files, `diff_addremove()` to feed created or deleted files, or `diff_unmerge()` to feed a file whose state is *unmerged* to the API. These are thin wrappers to a lower-level `diff_queue()` function that is flexible enough to record any of these kinds of changes.
- Once you finish feeding the pairs of files, call `diffcore_std()`. This will tell the diffcore library to go ahead and do its work.
- Calling `diff_flush()` will produce the output.

Data structures

- `struct diff_filespec`

This is the internal representation for a single file (blob). It records the blob object name (if known — for a work tree file it typically is a NUL SHA-1), filemode and pathname. This is what the `diff_addremove()`, `diff_change()` and `diff_unmerge()` synthesize and feed `diff_queue()` function with.

- `struct diff_filepair`

This records a pair of `struct diff_filespec`; the filespec for a file in the "old" set (i.e. preimage) is called `one`, and the filespec for a file in the "new" set (i.e. postimage) is called `two`. A change that represents file creation has NULL in `one`, and file deletion has NULL in `two`.

A `filepair` starts pointing at `one` and `two` that are from the same filename, but `diffcore_std()` can break pairs and match component filespecs with other filespecs from a different filepair to form new filepair. This is called *rename detection*.

- `struct diff_queue`

This is a collection of filepairs. Notable members are:

`queue`
: An array of pointers to `struct diff_filepair`. This dynamically grows as you add filepairs;

`alloc`
: The allocated size of the `queue` array;

`nr`
: The number of elements in the `queue` array.

    - `struct diff_options`

This describes the set of options the calling program wants to affect the operation of diffcore library with.

Notable members are:

`output_format`
: The output format used when `diff_flush()` is run.

`context`
: Number of context lines to generate in patch output.

`break_opt`, `detect_rename`, `rename-score`, `rename_limit`
: Affects the way detection logic for complete rewrites, renames and copies.

`abbrev`
: Number of hexdigits to abbreviate raw format output to.

`pickaxe`
: A constant string (can and typically does contain newlines to look for a block of text, not just a single line) to filter out the filepairs that do not change the number of strings contained in its preimage and postimage of the diff_queue.

`flags`
: This is mostly a collection of boolean options that affects the operation, but some do not have anything to do with the diffcore library.

`touched_flags`
: Records whether a flag has been changed due to user request (rather than just set/unset by default).

`set_default`
: Callback which allows tweaking the options in diff_setup_done().

  BINARY, TEXT
  : Affects the way how a file that is seemingly binary is treated.

  FULL_INDEX
  : Tells the patch output format not to use abbreviated object names on the "index" lines.

  FIND_COPIES_HARDER
  : Tells the diffcore library that the caller is feeding unchanged filepairs to allow copies from unmodified files be detected.

  COLOR_DIFF
  : Output should be colored.

  COLOR_DIFF_WORDS
  : Output is a colored word-diff.

  NO_INDEX
  : Tells diff-files that the input is not tracked files but files in random locations on the filesystem.

  ALLOW_EXTERNAL
  : Tells output routine that it is Ok to call user specified patch output routine. Plumbing disables this to ensure stable output.

  QUIET
  : Do not show any output.

  REVERSE_DIFF
  : Tells the library that the calling program is feeding the filepairs reversed; `one` is two, and `two` is one.

  EXIT_WITH_STATUS
  : For communication between the calling program and the options parser; tell the calling program to signal the presence of difference using program exit code.

  HAS_CHANGES
  : Internal; used for optimization to see if there is any change.

  SILENT_ON_REMOVE
  : Affects if diff-files shows removed files.

  RECURSIVE, TREE_IN_RECURSIVE
  : Tells if tree traversal done by tree-diff should recursively descend into a tree object pair that are different in preimage and postimage set.

(JC)

Last updated 2014-11-27 19:56:10 CET

directory listing API

The directory listing API is used to enumerate paths in the work tree, optionally taking `.git/info/exclude` and `.gitignore` files per directory into account.

Data structure

`struct dir_struct` structure is used to pass directory traversal options to the library and to record the paths discovered. A single `struct dir_struct` is used regardless of whether or not the traversal recursively descends into subdirectories.

The notable options are:

`exclude_per_dir`
: The name of the file to be read in each directory for excluded files (typically `.gitignore`).

`flags`
: A bit-field of options (the `*IGNORED*` flags are mutually exclusive):

    `DIR_SHOW_IGNORED`
    : Return just ignored files in `entries[]`, not untracked files.

    `DIR_SHOW_IGNORED_TOO`
    : Similar to `DIR_SHOW_IGNORED`, but return ignored files in `ignored[]` in addition to untracked files in `entries[]`.

    `DIR_COLLECT_IGNORED`
    : Special mode for git-add. Return ignored files in `ignored[]` and untracked files in `entries[]`. Only returns ignored files that match pathspec exactly (no wildcards). Does not recurse into ignored directories.

    `DIR_SHOW_OTHER_DIRECTORIES`
    : Include a directory that is not tracked.

    `DIR_HIDE_EMPTY_DIRECTORIES`
    : Do not include a directory that is not tracked and is empty.

    `DIR_NO_GITLINKS`
    : If set, recurse into a directory that looks like a Git directory. Otherwise it is shown as a directory.

The result of the enumeration is left in these fields:

`entries[]`
: An array of `struct dir_entry`, each element of which describes a path.

`nr`
: The number of members in `entries[]` array.

`alloc`
: Internal use; keeps track of allocation of `entries[]` array.

`ignored[]`
: An array of `struct dir_entry`, used for ignored paths with the `DIR_SHOW_IGNORED_TOO` and `DIR_COLLECT_IGNORED` flags.

`ignored_nr`
: The number of members in `ignored[]` array.

Calling sequence

Note: index may be looked at for .gitignore files that are CE_SKIP_WORKTREE marked. If you to exclude files, make sure you have loaded index first.

- Prepare `struct dir_struct dir` and clear it with `memset(&dir, 0, sizeof(dir))`.
- To add single exclude pattern, call `add_exclude_list()` and then `add_exclude()`.
- To add patterns from a file (e.g. `.git/info/exclude`), call `add_excludes_from_file()` , and/or set `dir.exclude_per_dir`. A short-hand function `setup_standard_excludes()` can be used to set up the standard set of exclude settings.
- Set options described in the Data Structure section above.
- Call `read_directory()`.
- Use `dir.entries[]`.
- Call `clear_directory()` when none of the contained elements are no longer in use.

(JC)

Last updated 2014-11-27 19:55:05 CET

Error reporting in git

`die`, `usage`, `error`, and `warning` report errors of various kinds.

- `die` is for fatal application errors. It prints a message to the user and exits with status 128.
- `usage` is for errors in command line usage. After printing its message, it exits with status 129. (See also `usage_with_options` in the parse-options API.)
- `error` is for non-fatal library errors. It prints a message to the user and returns -1 for convenience in signaling the error to the caller.
- `warning` is for reporting situations that probably should not occur but which the user (and Git) can continue to work around without running into too many problems. Like `error`, it returns -1 after reporting the situation to the caller.

Customizable error handlers

The default behavior of `die` and `error` is to write a message to stderr and then exit or return as appropriate. This behavior can be overridden using `set_die_routine` and `set_error_routine`. For example, "git daemon" uses set_die_routine to write the reason `die` was called to syslog before exiting.

Library errors

Functions return a negative integer on error. Details beyond that vary from function to function:

- Some functions return -1 for all errors. Others return a more specific value depending on how the caller might want to react to the error.
- Some functions report the error to stderr with `error`, while others leave that for the caller to do.
- errno is not meaningful on return from most functions (except for thin wrappers for system calls).

Check the function's API documentation to be sure.

Caller-handled errors

An increasing number of functions take a parameter *struct strbuf *err*. On error, such functions append a message about what went wrong to the *err* strbuf. The message is meant to be complete enough to be passed to `die` or `error` as-is. For example:

```
if (ref_transaction_commit(transaction, &err))
        die("%s", err.buf);
```

The *err* parameter will be untouched if no error occurred, so multiple function calls can be chained:

```
t = ref_transaction_begin(&err);
if (!t ||
    ref_transaction_update(t, "HEAD", ..., &err) ||
    ret_transaction_commit(t, &err))
        die("%s", err.buf);
```

The *err* parameter must be a pointer to a valid strbuf. To silence a message, pass a strbuf that is explicitly ignored:

```
if (thing_that_can_fail_in_an_ignorable_way(..., &err))
        /* This failure is okay. */
        strbuf_reset(&err);
```

Last updated 2015-05-03 21:16:24 CEST

gitattributes API

gitattributes mechanism gives a uniform way to associate various attributes to set of paths.

Data Structure

`struct git_attr`
: An attribute is an opaque object that is identified by its name. Pass the name to `git_attr()` function to obtain the object of this type. The internal representation of this structure is of no interest to the calling programs. The name of the attribute can be retrieved by calling `git_attr_name()`.

`struct git_attr_check`
: This structure represents a set of attributes to check in a call to `git_check_attr()` function, and receives the results.

Attribute Values

An attribute for a path can be in one of four states: Set, Unset, Unspecified or set to a string, and `.value` member of `struct git_attr_check` records it. There are three macros to check these:

`ATTR_TRUE()`
: Returns true if the attribute is Set for the path.

`ATTR_FALSE()`
: Returns true if the attribute is Unset for the path.

`ATTR_UNSET()`
: Returns true if the attribute is Unspecified for the path.

If none of the above returns true, `.value` member points at a string value of the attribute for the path.

Querying Specific Attributes

- Prepare an array of `struct git_attr_check` to define the list of attributes you would want to check. To populate this array, you would need to define necessary attributes by calling `git_attr()` function.
- Call `git_check_attr()` to check the attributes for the path.
- Inspect `git_attr_check` structure to see how each of the attribute in the array is defined for the path.

Example

To see how attributes "crlf" and "indent" are set for different paths.

1. Prepare an array of `struct git_attr_check` with two elements (because we are checking two attributes). Initialize their `attr` member with pointers to `struct git_attr` obtained by calling `git_attr()`:

```
static struct git_attr_check check[2];
static void setup_check(void)
{
        if (check[0].attr)
                return; /* already done */
        check[0].attr = git_attr("crlf");
        check[1].attr = git_attr("ident");
}
```

1. Call `git_check_attr()` with the prepared array of `struct git_attr_check`:

```
        const char *path;

        setup_check();
        git_check_attr(path, ARRAY_SIZE(check), check);
```

1. Act on `.value` member of the result, left in `check[]`:

```
        const char *value = check[0].value;

        if (ATTR_TRUE(value)) {
                The attribute is Set, by listing only the name of the
                attribute in the gitattributes file for the path.
        } else if (ATTR_FALSE(value)) {
                The attribute is Unset, by listing the name of the
                attribute prefixed with a dash - for the path.
        } else if (ATTR_UNSET(value)) {
                The attribute is neither set nor unset for the path.
        } else if (!strcmp(value, "input")) {
                If none of ATTR_TRUE(), ATTR_FALSE(), or ATTR_UNSET() is
                true, the value is a string set in the gitattributes
                file for the path by saying "attr=value".
        } else if (... other check using value as string ...) {
                ...
        }
```

Querying All Attributes

To get the values of all attributes associated with a file:

- Call `git_all_attrs()`, which returns an array of `git_attr_check` structures.
- Iterate over the `git_attr_check` array to examine the attribute names and values. The name of the attribute described by a `git_attr_check` object can be retrieved via `git_attr_name(check[i].attr)`. (Please note that no items will be returned for unset attributes, so `ATTR_UNSET()` will return false for all returned `git_array_check` objects.)
- Free the `git_array_check` array.

Last updated 2014-11-27 19:57:04 CET

grep API

Talk about <grep.h>, things like:

- grep_buffer()

(JC)

Last updated 2014-01-25 09:03:55 CET

hashmap API

The hashmap API is a generic implementation of hash-based key-value mappings.

Data Structures

`struct hashmap`
: The hash table structure. Members can be used as follows, but should not be modified directly:

  The `size` member keeps track of the total number of entries (0 means the hashmap is empty).

  `tablesize` is the allocated size of the hash table. A non-0 value indicates that the hashmap is initialized. It may also be useful for statistical purposes (i.e. `size / tablesize` is the current load factor).

  `cmpfn` stores the comparison function specified in `hashmap_init()`. In advanced scenarios, it may be useful to change this, e.g. to switch between case-sensitive and case-insensitive lookup.

`struct hashmap_entry`
: An opaque structure representing an entry in the hash table, which must be used as first member of user data structures. Ideally it should be followed by an int-sized member to prevent unused memory on 64-bit systems due to alignment.

  The `hash` member is the entry's hash code and the `next` member points to the next entry in case of collisions (i.e. if multiple entries map to the same bucket).

`struct hashmap_iter`
: An iterator structure, to be used with hashmap_iter_* functions.

Types

`int (*hashmap_cmp_fn)(const void *entry, const void *entry_or_key, const void *keydata)`

User-supplied function to test two hashmap entries for equality. Shall return 0 if the entries are equal.

This function is always called with non-NULL `entry` / `entry_or_key` parameters that have the same hash code. When looking up an entry, the `key` and `keydata` parameters to hashmap_get and hashmap_remove are always passed as second and third argument, respectively. Otherwise, `keydata` is NULL.

Functions

`unsigned int strhash(const char *buf)`

`unsigned int strihash(const char *buf)`

`unsigned int memhash(const void *buf, size_t len)`

`unsigned int memihash(const void *buf, size_t len)`

Ready-to-use hash functions for strings, using the FNV-1 algorithm (see http://www.isthe.com/chongo/tech/comp/fnv).

`strhash` and `strihash` take 0-terminated strings, while `memhash` and `memihash` operate on arbitrary-length memory.

`strihash` and `memihash` are case insensitive versions.

`unsigned int sha1hash(const unsigned char *sha1)`

Converts a cryptographic hash (e.g. SHA-1) into an int-sized hash code for use in hash tables. Cryptographic hashes are supposed to have uniform distribution, so in contrast to `memhash()`, this just copies the first `sizeof(int)` bytes without shuffling any bits. Note that the results will be different on big-endian and little-endian platforms, so they should not be stored or transferred over the net.

`void hashmap_init(struct hashmap *map, hashmap_cmp_fn equals_function, size_t initial_size)`

Initializes a hashmap structure.

`map` is the hashmap to initialize.

The `equals_function` can be specified to compare two entries for equality. If NULL, entries are considered equal if their hash codes are equal.

If the total number of entries is known in advance, the `initial_size` parameter may be used to preallocate a sufficiently large table and thus prevent expensive resizing. If 0, the table is dynamically resized.

`void hashmap_free(struct hashmap *map, int free_entries)`

Frees a hashmap structure and allocated memory.

`map` is the hashmap to free.

If `free_entries` is true, each hashmap_entry in the map is freed as well (using stdlib's free()).

`void hashmap_entry_init(void *entry, unsigned int hash)`

Initializes a hashmap_entry structure.

`entry` points to the entry to initialize.

`hash` is the hash code of the entry.

`void *hashmap_get(const struct hashmap *map, const void *key, const void *keydata)`

Returns the hashmap entry for the specified key, or NULL if not found.

`map` is the hashmap structure.

`key` is a hashmap_entry structure (or user data structure that starts with hashmap_entry) that has at least been initialized with the proper hash code (via `hashmap_entry_init`).

If an entry with matching hash code is found, `key` and `keydata` are passed to `hashmap_cmp_fn` to decide whether the entry matches the key.

`void *hashmap_get_from_hash(const struct hashmap *map, unsigned int hash, const void *keydata)`

Returns the hashmap entry for the specified hash code and key data, or NULL if not found.

`map` is the hashmap structure.

`hash` is the hash code of the entry to look up.

If an entry with matching hash code is found, `keydata` is passed to `hashmap_cmp_fn` to decide whether the entry matches the key. The `entry_or_key` parameter points to a bogus hashmap_entry structure that should not be used in the comparison.

`void *hashmap_get_next(const struct hashmap *map, const void *entry)`

Returns the next equal hashmap entry, or NULL if not found. This can be used to iterate over duplicate entries (see `hashmap_add`).

`map` is the hashmap structure.

`entry` is the hashmap_entry to start the search from, obtained via a previous call to `hashmap_get` or `hashmap_get_next`.

`void hashmap_add(struct hashmap *map, void *entry)`
: Adds a hashmap entry. This allows to add duplicate entries (i.e. separate values with the same key according to hashmap_cmp_fn).

  `map` is the hashmap structure.

  `entry` is the entry to add.

`void *hashmap_put(struct hashmap *map, void *entry)`
: Adds or replaces a hashmap entry. If the hashmap contains duplicate entries equal to the specified entry, only one of them will be replaced.

  `map` is the hashmap structure.

  `entry` is the entry to add or replace.

  Returns the replaced entry, or NULL if not found (i.e. the entry was added).

`void *hashmap_remove(struct hashmap *map, const void *key, const void *keydata)`
: Removes a hashmap entry matching the specified key. If the hashmap contains duplicate entries equal to the specified key, only one of them will be removed.

  `map` is the hashmap structure.

  `key` is a hashmap_entry structure (or user data structure that starts with hashmap_entry) that has at least been initialized with the proper hash code (via `hashmap_entry_init`).

  If an entry with matching hash code is found, `key` and `keydata` are passed to `hashmap_cmp_fn` to decide whether the entry matches the key.

  Returns the removed entry, or NULL if not found.

`void hashmap_iter_init(struct hashmap *map, struct hashmap_iter *iter)`

`void *hashmap_iter_next(struct hashmap_iter *iter)`

`void *hashmap_iter_first(struct hashmap *map, struct hashmap_iter *iter)`
: Used to iterate over all entries of a hashmap.

  `hashmap_iter_init` initializes a `hashmap_iter` structure.

  `hashmap_iter_next` returns the next hashmap_entry, or NULL if there are no more entries.

  `hashmap_iter_first` is a combination of both (i.e. initializes the iterator and returns the first entry, if any).

`const char *strintern(const char *string)`

`const void *memintern(const void *data, size_t len)`
: Returns the unique, interned version of the specified string or data, similar to the `String.intern` API in Java and .NET, respectively. Interned strings remain valid for the entire lifetime of the process.

  Can be used as `[x]strdup()` or `xmemdupz` replacement, except that interned strings / data must not be modified or freed.

  Interned strings are best used for short strings with high probability of duplicates.

  Uses a hashmap to store the pool of interned strings.

Usage example

Here's a simple usage example that maps long keys to double values.

```
struct hashmap map;

struct long2double {
        struct hashmap_entry ent; /* must be the first member! */
        long key;
        double value;
};

static int long2double_cmp(const struct long2double *e1, const struct long2double *e2, const void *unused)
{
        return !(e1->key == e2->key);
}

void long2double_init(void)
{
        hashmap_init(&map, (hashmap_cmp_fn) long2double_cmp, 0);
}

void long2double_free(void)
{
        hashmap_free(&map, 1);
}

static struct long2double *find_entry(long key)
{
        struct long2double k;
        hashmap_entry_init(&k, memhash(&key, sizeof(long)));
        k.key = key;
        return hashmap_get(&map, &k, NULL);
}

double get_value(long key)
{
        struct long2double *e = find_entry(key);
        return e ? e->value : 0;
}

void set_value(long key, double value)
{
        struct long2double *e = find_entry(key);
        if (!e) {
                e = malloc(sizeof(struct long2double));
                hashmap_entry_init(e, memhash(&key, sizeof(long)));
                e->key = key;
                hashmap_add(&map, e);
        }
        e->value = value;
}
```

Using variable-sized keys

The `hashmap_entry_get` and `hashmap_entry_remove` functions expect an ordinary `hashmap_entry` structure as key to find the correct entry. If the key data is variable-sized (e.g. a FLEX_ARRAY string) or quite large, it is undesirable to create a full-fledged entry structure on the heap and copy all the key data into the structure.

In this case, the `keydata` parameter can be used to pass variable-sized key data directly to the comparison function, and the `key` parameter can be a stripped-down, fixed size entry structure allocated on the stack.

See test-hashmap.c for an example using arbitrary-length strings as keys.

Last updated 2014-11-27 19:58:08 CET

history graph API

The graph API is used to draw a text-based representation of the commit history. The API generates the graph in a line-by-line fashion.

Functions

Core functions:

- `graph_init()` creates a new `struct git_graph`
- `graph_update()` moves the graph to a new commit.
- `graph_next_line()` outputs the next line of the graph into a strbuf. It does not add a terminating newline.
- `graph_padding_line()` outputs a line of vertical padding in the graph. It is similar to `graph_next_line()`, but is guaranteed to never print the line containing the current commit. Where `graph_next_line()` would print the commit line next, `graph_padding_line()` prints a line that simply extends all branch lines downwards one row, leaving their positions unchanged.
- `graph_is_commit_finished()` determines if the graph has output all lines necessary for the current commit. If `graph_update()` is called before all lines for the current commit have been printed, the next call to `graph_next_line()` will output an ellipsis, to indicate that a portion of the graph was omitted.

The following utility functions are wrappers around `graph_next_line()` and `graph_is_commit_finished()`. They always print the output to stdout. They can all be called with a NULL graph argument, in which case no graph output will be printed.

- `graph_show_commit()` calls `graph_next_line()` and `graph_is_commit_finished()` until one of them return non-zero. This prints all graph lines up to, and including, the line containing this commit. Output is printed to stdout. The last line printed does not contain a terminating newline.
- `graph_show_oneline()` calls `graph_next_line()` and prints the result to stdout. The line printed does not contain a terminating newline.
- `graph_show_padding()` calls `graph_padding_line()` and prints the result to stdout. The line printed does not contain a terminating newline.

- `graph_show_remainder()` calls `graph_next_line()` until `graph_is_commit_finished()` returns non-zero. Output is printed to stdout. The last line printed does not contain a terminating newline. Returns 1 if output was printed, and 0 if no output was necessary.
- `graph_show_strbuf()` prints the specified strbuf to stdout, prefixing all lines but the first with a graph line. The caller is responsible for ensuring graph output for the first line has already been printed to stdout. (This can be done with `graph_show_commit()` or `graph_show_oneline()`.) If a NULL graph is supplied, the strbuf is printed as-is.
- `graph_show_commit_msg()` is similar to `graph_show_strbuf()`, but it also prints the remainder of the graph, if more lines are needed after the strbuf ends. It is better than directly calling `graph_show_strbuf()` followed by `graph_show_remainder()` since it properly handles buffers that do not end in a terminating newline. The output printed by `graph_show_commit_msg()` will end in a newline if and only if the strbuf ends in a newline.

Data structure

`struct git_graph` is an opaque data type used to store the current graph state.

Calling sequence

- Create a `struct git_graph` by calling `graph_init()`. When using the revision walking API, this is done automatically by `setup_revisions()` if the *--graph* option is supplied.
- Use the revision walking API to walk through a group of contiguous commits. The `get_revision()` function automatically calls `graph_update()` each time it is invoked.
- For each commit, call `graph_next_line()` repeatedly, until `graph_is_commit_finished()` returns non-zero. Each call go `graph_next_line()` will output a single line of the graph. The resulting lines will not contain any newlines. `graph_next_line()` returns 1 if the resulting line contains the current commit, or 0 if this is merely a line needed to adjust the graph before or after the current commit. This return value can be used to determine where to print the commit summary information alongside the graph output.

Limitations

- `graph_update()` must be called with commits in topological order. It should not be called on a commit if it has already been invoked with an ancestor of that commit, or the graph output will be incorrect.
- `graph_update()` must be called on a contiguous group of commits. If `graph_update()` is called on a particular commit, it should later be called on all parents of that commit. Parents must not be skipped, or the graph output will appear incorrect.

  `graph_update()` may be used on a pruned set of commits only if the parent list has been rewritten so as to include only ancestors from the pruned set.
- The graph API does not currently support reverse commit ordering. In order to implement reverse ordering, the graphing API needs an (efficient) mechanism to find the children of a commit.

Sample usage

```
struct commit *commit;
struct git_graph *graph = graph_init(opts);

while ((commit = get_revision(opts)) != NULL) {
        graph_update(graph, commit);
        while (!graph_is_commit_finished(graph))
        {
                struct strbuf sb;
                int is_commit_line;

                strbuf_init(&sb, 0);
                is_commit_line = graph_next_line(graph, &sb);
                fputs(sb.buf, stdout);

                if (is_commit_line)
                        log_tree_commit(opts, commit);
                else
                        putchar(opts->diffopt.line_termination);
        }
}
```

Sample output

The following is an example of the output from the graph API. This output does not include any commit summary information—callers are responsible for outputting that information, if desired.

```
*
*
*
|\
* |
| | *
| \ \
|  \ \
*-. \ \
|\ \ \ \
| | * | |
| | | | | *
| | | | | *
| | | | | *
| | | | | |\
| | | | | | *
| * | | | | |
| | | | | *  \
| | | | | |\  |
| | | | * | | |
| | | | * | | |
* | | | | | | |
| |/ / / / / /
|/| / / / / /
* | | | | | |
|/ / / / / /
* | | | | |
| | | | | *
| | | | |/
| | | | *
```

Last updated 2014-11-27 19:54:14 CET

in-core index API

Talk about <read-cache.c> and <cache-tree.c>, things like:

- cache → the_index macros
- read_index()
- write_index()
- ie_match_stat() and ie_modified(); how they are different and when to use which.
- index_name_pos()
- remove_index_entry_at()
- remove_file_from_index()
- add_file_to_index()
- add_index_entry()
- refresh_index()
- discard_index()
- cache_tree_invalidate_path()
- cache_tree_update()

(JC, Linus)

Last updated 2014-01-25 09:03:55 CET

lockfile API

The lockfile API serves two purposes:

- Mutual exclusion and atomic file updates. When we want to change a file, we create a lockfile `<filename>.lock`, write the new file contents into it, and then rename the lockfile to its final destination `<filename>`. We create the `<filename>.lock` file with `O_CREAT|O_EXCL` so that we can notice and fail if somebody else has already locked the file, then atomically rename the lockfile to its final destination to commit the changes and unlock the file.
- Automatic cruft removal. If the program exits after we lock a file but before the changes have been committed, we want to make sure that we remove the lockfile. This is done by remembering the lockfiles we have created in a linked list and setting up an `atexit(3)` handler and a signal handler that clean up the lockfiles. This mechanism ensures that outstanding lockfiles are cleaned up if the program exits (including when `die()` is called) or if the program dies on a signal.

Please note that lockfiles only block other writers. Readers do not block, but they are guaranteed to see either the old contents of the file or the new contents of the file (assuming that the filesystem implements `rename(2)` atomically).

Calling sequence

The caller:

- Allocates a `struct lock_file` either as a static variable or on the heap, initialized to zeros. Once you use the structure to call the `hold_lock_file_*` family of functions, it belongs to the lockfile subsystem and its storage must remain valid throughout the life of the program (i.e. you cannot use an on-stack variable to hold this structure).
- Attempts to create a lockfile by passing that variable and the path of the final destination (e.g. `$GIT_DIR/index`) to `hold_lock_file_for_update` or `hold_lock_file_for_append`.
- Writes new content for the destination file by either:
- writing to the file descriptor returned by the `hold_lock_file_*` functions (also available via `lock->fd`).
- calling `fdopen_lock_file` to get a `FILE` pointer for the open file and writing to the file using stdio.

When finished writing, the caller can:

- Close the file descriptor and rename the lockfile to its final destination by calling `commit_lock_file` or `commit_lock_file_to`.
- Close the file descriptor and remove the lockfile by calling `rollback_lock_file`.
- Close the file descriptor without removing or renaming the lockfile by calling `close_lock_file`, and later call `commit_lock_file`, `commit_lock_file_to`, `rollback_lock_file`, or `reopen_lock_file`.

Even after the lockfile is committed or rolled back, the `lock_file` object must not be freed or altered by the caller. However, it may be reused; just pass it to another call of `hold_lock_file_for_update` or `hold_lock_file_for_append`.

If the program exits before you have called one of `commit_lock_file`, `commit_lock_file_to`, `rollback_lock_file`, or `close_lock_file`, an `atexit(3)` handler will close and remove the lockfile, rolling back any uncommitted changes.

If you need to close the file descriptor you obtained from a `hold_lock_file_*` function yourself, do so by calling `close_lock_file`. You should never call `close(2)` or `fclose(3)` yourself! Otherwise the `struct lock_file` structure would still think that the file descriptor needs to be closed, and a commit or rollback would result in duplicate calls to `close(2)`. Worse yet, if you close and then later open another file descriptor for a completely different purpose, then a commit or rollback might close that unrelated file descriptor.

Error handling

The `hold_lock_file_*` functions return a file descriptor on success or -1 on failure (unless `LOCK_DIE_ON_ERROR` is used; see below). On errors, `errno` describes the reason for failure. Errors can be reported by passing `errno` to one of the following helper functions:

unable_to_lock_message
: Append an appropriate error message to a `strbuf`.

unable_to_lock_error
: Emit an appropriate error message using `error()`.

unable_to_lock_die
: Emit an appropriate error message and `die()`.

Similarly, `commit_lock_file`, `commit_lock_file_to`, and `close_lock_file` return 0 on success. On failure they set `errno` appropriately, do their best to roll back the lockfile, and return -1.

Flags

The following flags can be passed to `hold_lock_file_for_update` or `hold_lock_file_for_append`:

LOCK_NO_DEREF
: Usually symbolic links in the destination path are resolved and the lockfile is created by adding ".lock" to the resolved path. If `LOCK_NO_DEREF` is set, then the lockfile is created by adding ".lock" to the path argument itself. This option is used, for example, when locking a symbolic reference, which for backwards-compatibility reasons can be a symbolic link containing the name of the referred-to-reference.

LOCK_DIE_ON_ERROR
: If a lock is already taken for the file, `die()` with an error message. If this option is not specified, trying to lock a file that is already locked returns -1 to the caller.

The functions

hold_lock_file_for_update
: Take a pointer to `struct lock_file`, the path of the file to be locked (e.g. `$GIT_DIR/index`) and a flags argument (see above). Attempt to create a lockfile for the destination and return the file descriptor for writing to the file.

hold_lock_file_for_append
: Like `hold_lock_file_for_update`, but before returning copy the existing contents of the file (if any) to the lockfile and position its write pointer at the end of the file.

fdopen_lock_file
: Associate a stdio stream with the lockfile. Return NULL (**without** rolling back the lockfile) on error. The stream is closed automatically when `close_lock_file` is called or when the file is committed or rolled back.

get_locked_file_path
: Return the path of the file that is locked by the specified lock_file object. The caller must free the memory.

commit_lock_file
: Take a pointer to the `struct lock_file` initialized with an earlier call to `hold_lock_file_for_update` or `hold_lock_file_for_append`, close the file descriptor, and rename the lockfile to its final destination. Return 0 upon success. On failure, roll back the lock file and return -1, with `errno` set to the value from the failing call to `close(2)` or `rename(2)`. It is a bug to call `commit_lock_file` for a `lock_file` object that is not currently locked.

commit_lock_file_to
: Like `commit_lock_file()`, except that it takes an explicit `path` argument to which the lockfile should be renamed. The `path` must be on the same filesystem as the lock file.

rollback_lock_file
: Take a pointer to the `struct lock_file` initialized with an earlier call to `hold_lock_file_for_update` or `hold_lock_file_for_append`, close the file descriptor and remove the lockfile. It is a NOOP to call `rollback_lock_file()` for a `lock_file` object that has already been committed or rolled back.

close_lock_file
: Take a pointer to the `struct lock_file` initialized with an earlier call to `hold_lock_file_for_update` or `hold_lock_file_for_append`. Close the file descriptor (and the file pointer if it has been opened using `fdopen_lock_file`). Return 0 upon success. On failure to `close(2)`, return a negative value and roll back the lock file. Usually `commit_lock_file`, `commit_lock_file_to`, or `rollback_lock_file` should eventually be called if `close_lock_file` succeeds.

reopen_lock_file
: Re-open a lockfile that has been closed (using `close_lock_file`) but not yet committed or rolled back. This can be used to implement a sequence of operations like the following:

  - Lock file.
  - Write new contents to lockfile, then `close_lock_file` to cause the contents to be written to disk.
  - Pass the name of the lockfile to another program to allow it (and nobody else) to inspect the contents you wrote, while still holding the lock yourself.
  - `reopen_lock_file` to reopen the lockfile. Make further updates to the contents.
  - `commit_lock_file` to make the final version permanent.

Last updated 2014-11-27 19:58:08 CET

merge API

The merge API helps a program to reconcile two competing sets of improvements to some files (e.g., unregistered changes from the work tree versus changes involved in switching to a new branch), reporting conflicts if found. The library called through this API is responsible for a few things.

- determining which trees to merge (recursive ancestor consolidation);
- lining up corresponding files in the trees to be merged (rename detection, subtree shifting), reporting edge cases like add/add and rename/rename conflicts to the user;
- performing a three-way merge of corresponding files, taking path-specific merge drivers (specified in `.gitattributes`) into account.

Data structures

- `mmbuffer_t`, `mmfile_t`

These store data usable for use by the xdiff backend, for writing and for reading, respectively. See `xdiff/xdiff.h` for the definitions and `diff.c` for examples.

- `struct ll_merge_options`

This describes the set of options the calling program wants to affect the operation of a low-level (single file) merge. Some options:

`virtual_ancestor`
: Behave as though this were part of a merge between common ancestors in a recursive merge. If a helper program is specified by the `[merge "<driver>"] recursive` configuration, it will be used (see gitattributes(5)).

`variant`
: Resolve local conflicts automatically in favor of one side or the other (as in *git merge-file* `--ours`/`--theirs`/`--union`). Can be `0`, `XDL_MERGE_FAVOR_OURS`, `XDL_MERGE_FAVOR_THEIRS`, or `XDL_MERGE_FAVOR_UNION`.

`renormalize`
: Resmudge and clean the "base", "theirs" and "ours" files before merging. Use this when the merge is likely to have overlapped with a change in smudge/clean or end-of-line normalization rules.

Low-level (single file) merge

`ll_merge`
: Perform a three-way single-file merge in core. This is a thin wrapper around `xdl_merge` that takes the path and any merge backend specified in `.gitattributes` or `.git/info/attributes` into account. Returns 0 for a clean merge.

Calling sequence:

- Prepare a `struct ll_merge_options` to record options. If you have no special requests, skip this and pass `NULL` as the `opts` parameter to use the default options.
- Allocate an mmbuffer_t variable for the result.
- Allocate and fill variables with the file's original content and two modified versions (using `read_mmfile`, for example).
- Call `ll_merge()`.
- Read the merged content from `result_buf.ptr` and `result_buf.size`.
- Release buffers when finished. A simple `free(ancestor.ptr); free(ours.ptr); free(theirs.ptr); free(result_buf.ptr);` will do.

If the modifications do not merge cleanly, `ll_merge` will return a nonzero value and `result_buf` will generally include a description of the conflict bracketed by markers such as the traditional `<<<<<<<` and `>>>>>>>`.

The `ancestor_label`, `our_label`, and `their_label` parameters are used to label the different sides of a conflict if the merge driver supports this.

Everything else

Talk about <merge-recursive.h> and merge_file():

- merge_trees() to merge with rename detection
- merge_recursive() for ancestor consolidation
- try_merge_command() for other strategies
- conflict format
- merge options

(Daniel, Miklos, Stephan, JC)

object access API

Talk about <sha1_file.c> and <object.h> family, things like

- read_sha1_file()
- read_object_with_reference()
- has_sha1_file()
- write_sha1_file()
- pretend_sha1_file()
- lookup_{object,commit,tag,blob,tree}
- parse_{object,commit,tag,blob,tree}
- Use of object flags

(JC, Shawn, Daniel, Dscho, Linus)

parse-options API

The parse-options API is used to parse and massage options in Git and to provide a usage help with consistent look.

Basics

The argument vector `argv[]` may usually contain mandatory or optional *non-option arguments*, e.g. a filename or a branch, and *options*. Options are optional arguments that start with a dash and that allow to change the behavior of a command.

- There are basically three types of options: *boolean* options, options with (mandatory) *arguments* and options with *optional arguments* (i.e. a boolean option that can be adjusted).
- There are basically two forms of options: *Short options* consist of one dash (`-`) and one alphanumeric character. *Long options* begin with two dashes (`--`) and some alphanumeric characters.
- Options are case-sensitive. Please define *lower-case long options* only.

The parse-options API allows:

- *stuck* and *separate form* of options with arguments. `-oArg` is stuck, `-o Arg` is separate form. `--option=Arg` is stuck, `--option Arg` is separate form.
- Long options may be *abbreviated*, as long as the abbreviation is unambiguous.
- Short options may be bundled, e.g. `-a -b` can be specified as `-ab`.
- Boolean long options can be *negated* (or *unset*) by prepending `no-`, e.g. `--no-abbrev` instead of `--abbrev`. Conversely, options that begin with `no-` can be *negated* by removing it. Other long options can be unset (e.g., set string to NULL, set integer to 0) by prepending `no-`.
- Options and non-option arguments can clearly be separated using the `--` option, e.g. `-a -b --option -- --this-is-a-file` indicates that `--this-is-a-file` must not be processed as an option.

Steps to parse options

1. `#include "parse-options.h"`
2. define a NULL-terminated `static const char * const builtin_foo_usage[]` array containing alternative usage strings
3. define `builtin_foo_options` array as described below in section *Data Structure*.

4. in `cmd_foo(int argc, const char **argv, const char *prefix)` call

   ```
   argc = parse_options(argc, argv, prefix, builtin_foo_options, builtin_foo_usage, flags);
   ```

   `parse_options()` will filter out the processed options of `argv[]` and leave the non-option arguments in `argv[]`. `argc` is updated appropriately because of the assignment.

   You can also pass NULL instead of a usage array as the fifth parameter of parse_options(), to avoid displaying a help screen with usage info and option list. This should only be done if necessary, e.g. to implement a limited parser for only a subset of the options that needs to be run before the full parser, which in turn shows the full help message.

   Flags are the bitwise-or of:

   `PARSE_OPT_KEEP_DASHDASH`
   : Keep the `--` that usually separates options from non-option arguments.

   `PARSE_OPT_STOP_AT_NON_OPTION`
   : Usually the whole argument vector is massaged and reordered. Using this flag, processing is stopped at the first non-option argument.

   `PARSE_OPT_KEEP_ARGV0`
   : Keep the first argument, which contains the program name. It's removed from argv[] by default.

   `PARSE_OPT_KEEP_UNKNOWN`
   : Keep unknown arguments instead of erroring out. This doesn't work for all combinations of arguments as users might expect it to do. E.g. if the first argument in `--unknown --known` takes a value (which we can't know), the second one is mistakenly interpreted as a known option. Similarly, if `PARSE_OPT_STOP_AT_NON_OPTION` is set, the second argument in `--unknown value` will be mistakenly interpreted as a non-option, not as a value belonging to the unknown option, the parser early. That's why parse_options() errors out if both options are set.

   `PARSE_OPT_NO_INTERNAL_HELP`
   : By default, parse_options() handles `-h`, `--help` and `--help-all` internally, by showing a help screen. This option turns it off and allows one to add custom handlers for these options, or to just leave them unknown.

Data Structure

The main data structure is an array of the `option` struct, say `static struct option builtin_add_options[]`. There are some macros to easily define options:

`OPT__ABBREV(&int_var)`
: Add `--abbrev[=<n>]`.

`OPT__COLOR(&int_var, description)`
: Add `--color[=<when>]` and `--no-color`.

`OPT__DRY_RUN(&int_var, description)`
: Add `-n, --dry-run`.

`OPT__FORCE(&int_var, description)`
: Add `-f, --force`.

`OPT__QUIET(&int_var, description)`
: Add `-q, --quiet`.

`OPT__VERBOSE(&int_var, description)`
: Add `-v, --verbose`.

`OPT_GROUP(description)`
: Start an option group. `description` is a short string that describes the group or an empty string. Start the description with an upper-case letter.

`OPT_BOOL(short, long, &int_var, description)`
: Introduce a boolean option. `int_var` is set to one with `--option` and set to zero with `--no-option`.

`OPT_COUNTUP(short, long, &int_var, description)`
: Introduce a count-up option. `int_var` is incremented on each use of `--option`, and reset to zero with `--no-option`.

`OPT_BIT(short, long, &int_var, description, mask)`
: Introduce a boolean option. If used, `int_var` is bitwise-ored with `mask`.

`OPT_NEGBIT(short, long, &int_var, description, mask)`
: Introduce a boolean option. If used, `int_var` is bitwise-anded with the inverted `mask`.

`OPT_SET_INT(short, long, &int_var, description, integer)`
: Introduce an integer option. `int_var` is set to `integer` with `--option`, and reset to zero with `--no-option`.

`OPT_STRING(short, long, &str_var, arg_str, description)`
: Introduce an option with string argument. The string argument is put into `str_var`.

`OPT_INTEGER(short, long, &int_var, description)`
: Introduce an option with integer argument. The integer is put into `int_var`.

`OPT_DATE(short, long, &int_var, description)`
: Introduce an option with date argument, see `approxidate()`. The timestamp is put into `int_var`.

`OPT_EXPIRY_DATE(short, long, &int_var, description)`
: Introduce an option with expiry date argument, see `parse_expiry_date()`. The timestamp is put into `int_var`.

`OPT_CALLBACK(short, long, &var, arg_str, description, func_ptr)`
: Introduce an option with argument. The argument will be fed into the function given by `func_ptr` and the result will be put into `var`. See *Option Callbacks* below for a more elaborate description.

`OPT_FILENAME(short, long, &var, description)`
: Introduce an option with a filename argument. The filename will be prefixed by passing the filename along with the prefix argument of `parse_options()` to `prefix_filename()`.

`OPT_ARGUMENT(long, description)`
: Introduce a long-option argument that will be kept in `argv[]`.

`OPT_NUMBER_CALLBACK(&var, description, func_ptr)`
: Recognize numerical options like -123 and feed the integer as if it was an argument to the function given by `func_ptr`. The result will be put into `var`. There can be only one such option definition. It cannot be negated and it takes no arguments. Short options that happen to be digits take precedence over it.

`OPT_COLOR_FLAG(short, long, &int_var, description)`
: Introduce an option that takes an optional argument that can have one of three values: "always", "never", or "auto". If the argument is not given, it defaults to "always". The `--no-` form works like `--long=never`; it cannot take an argument. If "always", set `int_var` to 1; if "never", set `int_var` to 0; if "auto", set `int_var` to 1 if stdout is a tty or a pager, 0 otherwise.

`OPT_NOOP_NOARG(short, long)`
: Introduce an option that has no effect and takes no arguments. Use it to hide deprecated options that are still to be recognized and ignored silently.

The last element of the array must be `OPT_END()`.

If not stated otherwise, interpret the arguments as follows:

- `short` is a character for the short option (e.g. `'e'` for `-e`, use `0` to omit),
- `long` is a string for the long option (e.g. `"example"` for `--example`, use `NULL` to omit),
- `int_var` is an integer variable,
- `str_var` is a string variable (`char *`),
- `arg_str` is the string that is shown as argument (e.g. `"branch"` will result in `<branch>`). If set to `NULL`, three dots (`...`) will be displayed.
- `description` is a short string to describe the effect of the option. It shall begin with a lower-case letter and a full stop (`.`) shall be omitted at the end.

Option Callbacks

The function must be defined in this form:

```
int func(const struct option *opt, const char *arg, int unset)
```

The callback mechanism is as follows:

- Inside `func`, the only interesting member of the structure given by `opt` is the void pointer `opt->value`. `*opt->value` will be the value that is saved into `var`, if you use `OPT_CALLBACK()`. For example, do `*(unsigned long *)opt->value = 42;` to get 42 into an `unsigned long` variable.
- Return value `0` indicates success and non-zero return value will invoke `usage_with_options()` and, thus, die.
- If the user negates the option, `arg` is `NULL` and `unset` is 1.

Sophisticated option parsing

If you need, for example, option callbacks with optional arguments or without arguments at all, or if you need other special cases, that are not handled by the macros above, you need to specify the members of the `option` structure manually.

This is not covered in this document, but well documented in `parse-options.h` itself.

Examples

See `test-parse-options.c` and `builtin/add.c`, `builtin/clone.c`, `builtin/commit.c`, `builtin/fetch.c`, `builtin/fsck.c`, `builtin/rm.c` for real-world examples.

Last updated 2014-11-27 19:57:04 CET

quote API

Talk about <quote.h>, things like

- sq_quote and unquote
- c_style quote and unquote
- quoting for foreign languages

(JC)

Last updated 2014-01-25 09:03:55 CET

ref iteration API

Iteration of refs is done by using an iterate function which will call a callback function for every ref. The callback function has this signature:

```
int handle_one_ref(const char *refname, const unsigned char *sha1,
                   int flags, void *cb_data);
```

There are different kinds of iterate functions which all take a callback of this type. The callback is then called for each found ref until the callback returns nonzero. The returned value is then also returned by the iterate function.

Iteration functions

- `head_ref()` just iterates the head ref.
- `for_each_ref()` iterates all refs.
- `for_each_ref_in()` iterates all refs which have a defined prefix and strips that prefix from the passed variable refname.
- `for_each_tag_ref()`, `for_each_branch_ref()`, `for_each_remote_ref()`, `for_each_replace_ref()` iterate refs from the respective area.
- `for_each_glob_ref()` iterates all refs that match the specified glob pattern.
- `for_each_glob_ref_in()` the previous and `for_each_ref_in()` combined.
- `head_ref_submodule()`, `for_each_ref_submodule()`, `for_each_ref_in_submodule()`, `for_each_tag_ref_submodule()`, `for_each_branch_ref_submodule()`, `for_each_remote_ref_submodule()` do the same as the functions described above but for a specified submodule.
- `for_each_rawref()` can be used to learn about broken ref and symref.
- `for_each_reflog()` iterates each reflog file.

Submodules

If you want to iterate the refs of a submodule you first need to add the submodules object database. You can do this by a code-snippet like this:

```
const char *path = "path/to/submodule"
if (add_submodule_odb(path))
        die("Error submodule '%s' not populated.", path);
```

add_submodule_odb() will return zero on success. If you do not do this you will get an error for each ref that it does not point to a valid object.

Note: As a side-effect of this you can not safely assume that all objects you lookup are available in superproject. All submodule objects will be available the same way as the superprojects objects.

Example:

```
static int handle_remote_ref(const char *refname,
                const unsigned char *sha1, int flags, void *cb_data)
{
        struct strbuf *output = cb_data;
        strbuf_addf(output, "%s\n", refname);
        return 0;
}

...

        struct strbuf output = STRBUF_INIT;
        for_each_remote_ref(handle_remote_ref, &output);
        printf("%s", output.buf);
```

Last updated 2014-11-27 19:56:10 CET

Remotes configuration API

The API in remote.h gives access to the configuration related to remotes. It handles all three configuration mechanisms historically and currently used by Git, and presents the information in a uniform fashion. Note that the code also handles plain URLs without any configuration, giving them just the default information.

struct remote

`name`
: The user's nickname for the remote

`url`
: An array of all of the url_nr URLs configured for the remote

`pushurl`
: An array of all of the pushurl_nr push URLs configured for the remote

`push`
: An array of refspecs configured for pushing, with push_refspec being the literal strings, and push_refspec_nr being the quantity.

`fetch`
: An array of refspecs configured for fetching, with fetch_refspec being the literal strings, and fetch_refspec_nr being the quantity.

`fetch_tags`
: The setting for whether to fetch tags (as a separate rule from the configured refspecs); -1 means never to fetch tags, 0 means to auto-follow tags based on the default heuristic, 1 means to always auto-follow tags, and 2 means to fetch all tags.

`receivepack`, `uploadpack`
: The configured helper programs to run on the remote side, for Git-native protocols.

`http_proxy`
: The proxy to use for curl (http, https, ftp, etc.) URLs.

struct remotes can be found by name with remote_get(), and iterated through with for_each_remote(). remote_get(NULL) will return the default remote, given the current branch and configuration.

struct refspec

A struct refspec holds the parsed interpretation of a refspec. If it will force updates (starts with a +), force is true. If it is a pattern (sides end with **) pattern is true. src and dest are the two sides (including** characters if present); if there is only one side, it is src, and dst is NULL; if sides exist but are empty (i.e., the refspec either starts or ends with `:`), the corresponding side is "".

An array of strings can be parsed into an array of struct refspecs using parse_fetch_refspec() or parse_push_refspec().

remote_find_tracking(), given a remote and a struct refspec with either src or dst filled out, will fill out the other such that the result is in the "fetch" specification for the remote (note that this evaluates patterns and returns a single result).

struct branch

Note that this may end up moving to branch.h

struct branch holds the configuration for a branch. It can be looked branch_get(NULL) for HEAD.

It contains:

`name`
: The short name of the branch.

`refname`
: The full path for the branch ref.

`remote_name`
: The name of the remote listed in the configuration.

`remote`
: The struct remote for that remote.

`merge_name`
: An array of the "merge" lines in the configuration.

`merge`
: An array of the struct refspecs used for the merge lines. That is, merge[i]→dst is a local tracking ref which should be merged into this branch by default.

`merge_nr`
: The number of merge configurations

branch_has_merge_config() returns true if the given branch has merge configuration given.

Other stuff

There is other stuff in remote.h that is related, in general, to the process of interacting with remotes.

(Daniel Barkalow)

Last updated 2014-11-27 19:56:10 CET

revision walking API

The revision walking API offers functions to build a list of revisions and then iterate over that list.

Calling sequence

The walking API has a given calling sequence: first you need to initialize a rev_info structure, then add revisions to control what kind of revision list do you want to get, finally you can iterate over the revision list.

Functions

`init_revisions`
: Initialize a rev_info structure with default values. The second parameter may be NULL or can be prefix path, and then the `.prefix` variable will be set to it. This is typically the first function you want to call when you want

to deal with a revision list. After calling this function, you are free to customize options, like set `.ignore_merges` to 0 if you don't want to ignore merges, and so on. See `revision.h` for a complete list of available options.

`add_pending_object`
: This function can be used if you want to add commit objects as revision information. You can use the `UNINTERESTING` object flag to indicate if you want to include or exclude the given commit (and commits reachable from the given commit) from the revision list.

> **Note** If you have the commits as a string list then you probably want to use setup_revisions(), instead of parsing each string and using this function.

`setup_revisions`
: Parse revision information, filling in the `rev_info` structure, and removing the used arguments from the argument list. Returns the number of arguments left that weren't recognized, which are also moved to the head of the argument list. The last parameter is used in case no parameter given by the first two arguments.

`prepare_revision_walk`
: Prepares the rev_info structure for a walk. You should check if it returns any error (non-zero return code) and if it does not, you can start using get_revision() to do the iteration.

`get_revision`
: Takes a pointer to a `rev_info` structure and iterates over it, returning a `struct commit *` each time you call it. The end of the revision list is indicated by returning a NULL pointer.

`reset_revision_walk`
: Reset the flags used by the revision walking api. You can use this to do multiple sequential revision walks.

Data structures

Talk about <revision.h>, things like:

- two diff_options, one for path limiting, another for output;
- remaining functions;

(Linus, JC, Dscho)

Last updated 2014-11-27 19:56:10 CET

run-command API

The run-command API offers a versatile tool to run sub-processes with redirected input and output as well as with a modified environment and an alternate current directory.

A similar API offers the capability to run a function asynchronously, which is primarily used to capture the output that the function produces in the caller in order to process it.

Functions

`child_process_init`
: Initialize a struct child_process variable.

`start_command`
: Start a sub-process. Takes a pointer to a `struct child_process` that specifies the details and returns pipe FDs (if requested). See below for details.

`finish_command`
: Wait for the completion of a sub-process that was started with start_command().

`run_command`
: A convenience function that encapsulates a sequence of start_command() followed by finish_command(). Takes a pointer to a `struct child_process` that specifies the details.

`run_command_v_opt`, `run_command_v_opt_cd_env`
: Convenience functions that encapsulate a sequence of start_command() followed by finish_command(). The argument argv specifies the program and its arguments. The argument opt is zero or more of the flags

RUN_COMMAND_NO_STDIN, RUN_GIT_CMD, RUN_COMMAND_STDOUT_TO_STDERR, or RUN_SILENT_EXEC_FAILURE that correspond to the members .no_stdin, .git_cmd, .stdout_to_stderr, .silent_exec_failure of struct child_process. The argument dir corresponds the member .dir. The argument env corresponds to the member .env.

The functions above do the following:

1. If a system call failed, errno is set and -1 is returned. A diagnostic is printed.
2. If the program was not found, then -1 is returned and errno is set to ENOENT; a diagnostic is printed only if .silent_exec_failure is 0.
3. Otherwise, the program is run. If it terminates regularly, its exit code is returned. No diagnostic is printed, even if the exit code is non-zero.
4. If the program terminated due to a signal, then the return value is the signal number + 128, ie. the same value that a POSIX shell's $? would report. A diagnostic is printed.

`start_async`
: Run a function asynchronously. Takes a pointer to a `struct async` that specifies the details and returns a set of pipe FDs for communication with the function. See below for details.

`finish_async`
: Wait for the completion of an asynchronous function that was started with start_async().

`run_hook`
: Run a hook. The first argument is a pathname to an index file, or NULL if the hook uses the default index file or no index is needed. The second argument is the name of the hook. The further arguments correspond to the hook arguments. The last argument has to be NULL to terminate the arguments list. If the hook does not exist or is not executable, the return value will be zero. If it is executable, the hook will be executed and the exit status of the hook is returned. On execution, .stdout_to_stderr and .no_stdin will be set. (See below.)

Data structures

- `struct child_process`

This describes the arguments, redirections, and environment of a command to run in a sub-process.

The caller:

1. allocates and clears (using child_process_init() or CHILD_PROCESS_INIT) a struct child_process variable;
2. initializes the members;
3. calls start_command();
4. processes the data;
5. closes file descriptors (if necessary; see below);
6. calls finish_command().

The .argv member is set up as an array of string pointers (NULL terminated), of which .argv[0] is the program name to run (usually without a path). If the command to run is a git command, set argv[0] to the command name without the *git-* prefix and set .git_cmd = 1.

Note that the ownership of the memory pointed to by .argv stays with the caller, but it should survive until `finish_command` completes. If the .argv member is NULL, `start_command` will point it at the .args `argv_array` (so you may use one or the other, but you must use exactly one). The memory in .args will be cleaned up automatically during `finish_command` (or during `start_command` when it is unsuccessful).

The members .in, .out, .err are used to redirect stdin, stdout, stderr as follows:

1. Specify 0 to request no special redirection. No new file descriptor is allocated. The child process simply inherits the channel from the parent.
2. Specify -1 to have a pipe allocated; start_command() replaces -1 by the pipe FD in the following way:

```
.in: Returns the writable pipe end into which the caller writes;
        the readable end of the pipe becomes the child's stdin.


.out, .err: Returns the readable pipe end from which the caller
        reads; the writable end of the pipe end becomes child's
        stdout/stderr.


The caller of start_command() must close the so returned FDs
after it has completed reading from/writing to it!
```

3. Specify a file descriptor > 0 to be used by the child:

```
.in: The FD must be readable; it becomes child's stdin.
.out: The FD must be writable; it becomes child's stdout.
.err: The FD must be writable; it becomes child's stderr.
```

```
The specified FD is closed by start_command(), even if it fails to
run the sub-process!
```

4. Special forms of redirection are available by setting these members to 1:

```
.no_stdin, .no_stdout, .no_stderr: The respective channel is
        redirected to /dev/null.


.stdout_to_stderr: stdout of the child is redirected to its
        stderr. This happens after stderr is itself redirected.
        So stdout will follow stderr to wherever it is
        redirected.
```

To modify the environment of the sub-process, specify an array of string pointers (NULL terminated) in .env:

1. If the string is of the form "VAR=value", i.e. it contains = the variable is added to the child process's environment.
2. If the string does not contain =, it names an environment variable that will be removed from the child process's environment.

If the .env member is NULL, `start_command` will point it at the .env_array `argv_array` (so you may use one or the other, but not both). The memory in .env_array will be cleaned up automatically during `finish_command` (or during `start_command` when it is unsuccessful).

To specify a new initial working directory for the sub-process, specify it in the .dir member.

If the program cannot be found, the functions return -1 and set errno to ENOENT. Normally, an error message is printed, but if .silent_exec_failure is set to 1, no message is printed for this special error condition.

- `struct async`

This describes a function to run asynchronously, whose purpose is to produce output that the caller reads.

The caller:

1. allocates and clears (memset(&asy, 0, sizeof(asy));) a struct async variable;
2. initializes .proc and .data;
3. calls start_async();
4. processes communicates with proc through .in and .out;
5. closes .in and .out;
6. calls finish_async().

The members .in, .out are used to provide a set of fd's for communication between the caller and the callee as follows:

1. Specify 0 to have no file descriptor passed. The callee will receive -1 in the corresponding argument.
2. Specify < 0 to have a pipe allocated; start_async() replaces with the pipe FD in the following way:

```
.in: Returns the writable pipe end into which the caller
writes; the readable end of the pipe becomes the function's
in argument.


.out: Returns the readable pipe end from which the caller
reads; the writable end of the pipe becomes the function's
out argument.


The caller of start_async() must close the returned FDs after it
has completed reading from/writing from them.
```

3. Specify a file descriptor > 0 to be used by the function:

```
.in: The FD must be readable; it becomes the function's in.
.out: The FD must be writable; it becomes the function's out.


The specified FD is closed by start_async(), even if it fails to
run the function.
```

The function pointer in .proc has the following signature:

```
int proc(int in, int out, void *data);
```

1. in, out specifies a set of file descriptors to which the function must read/write the data that it needs/produces. The function **must** close these descriptors before it returns. A descriptor may be -1 if the caller did not configure a descriptor for that direction.
2. data is the value that the caller has specified in the .data member of struct async.
3. The return value of the function is 0 on success and non-zero on failure. If the function indicates failure, finish_async() will report failure as well.

There are serious restrictions on what the asynchronous function can do because this facility is implemented by a thread in the same address space on most platforms (when pthreads is available), but by a pipe to a forked process otherwise:

1. It cannot change the program's state (global variables, environment, etc.) in a way that the caller notices; in other words, .in and .out are the only communication channels to the caller.
2. It must not change the program's state that the caller of the facility also uses.

Last updated 2014-11-27 19:58:08 CET

setup API

Talk about

- setup_git_directory()
- setup_git_directory_gently()
- is_inside_git_dir()
- is_inside_work_tree()
- setup_work_tree()

(Dscho)

Pathspec

See glossary-context.txt for the syntax of pathspec. In memory, a pathspec set is represented by "struct pathspec" and is prepared by parse_pathspec(). This function takes several arguments:

- magic_mask specifies what features that are NOT supported by the following code. If a user attempts to use such a feature, parse_pathspec() can reject it early.
- flags specifies other things that the caller wants parse_pathspec to perform.
- prefix and args come from cmd_* functions

get_pathspec() is obsolete and should never be used in new code.

parse_pathspec() helps catch unsupported features and reject them politely. At a lower level, different pathspec-related functions may not support the same set of features. Such pathspec-sensitive functions are guarded with GUARD_PATHSPEC(), which will die in an unfriendly way when an unsupported feature is requested.

The command designers are supposed to make sure that GUARD_PATHSPEC() never dies. They have to make sure all unsupported features are caught by parse_pathspec(), not by GUARD_PATHSPEC. grepping GUARD_PATHSPEC() should give the designers all pathspec-sensitive codepaths and what features they support.

A similar process is applied when a new pathspec magic is added. The designer lifts the GUARD_PATHSPEC restriction in the functions that support the new magic. At the same time (s)he has to make sure this new feature will be caught at parse_pathspec() in commands that cannot handle the new magic in some cases. grepping parse_pathspec() should help.

Last updated 2014-11-27 19:56:10 CET

sha1-array API

The sha1-array API provides storage and manipulation of sets of SHA-1 identifiers. The emphasis is on storage and processing efficiency, making them suitable for large lists. Note that the ordering of items is not preserved over some operations.

Data Structures

`struct sha1_array`
: A single array of SHA-1 hashes. This should be initialized by assignment from `SHA1_ARRAY_INIT`. The `sha1` member contains the actual data. The `nr` member contains the number of items in the set. The `alloc` and `sorted` members are used internally, and should not be needed by API callers.

Functions

`sha1_array_append`
: Add an item to the set. The sha1 will be placed at the end of the array (but note that some operations below may lose this ordering).

`sha1_array_lookup`
: Perform a binary search of the array for a specific sha1. If found, returns the offset (in number of elements) of the sha1. If not found, returns a negative integer. If the array is not sorted, this function has the side effect of sorting it.

`sha1_array_clear`
: Free all memory associated with the array and return it to the initial, empty state.

`sha1_array_for_each_unique`
: Efficiently iterate over each unique element of the list, executing the callback function for each one. If the array is not sorted, this function has the side effect of sorting it.

Examples

```
void print_callback(const unsigned char sha1[20],
                    void *data)
{
        printf("%s\n", sha1_to_hex(sha1));
}

void some_func(void)
{
        struct sha1_array hashes = SHA1_ARRAY_INIT;
        unsigned char sha1[20];

        /* Read objects into our set */
        while (read_object_from_stdin(sha1))
                sha1_array_append(&hashes, sha1);

        /* Check if some objects are in our set */
        while (read_object_from_stdin(sha1)) {
                if (sha1_array_lookup(&hashes, sha1) >= 0)
                        printf("it's in there!\n");

        /*
         * Print the unique set of objects. We could also have
         * avoided adding duplicate objects in the first place,
         * but we would end up re-sorting the array repeatedly.
         * Instead, this will sort once and then skip duplicates
         * in linear time.
         */
        sha1_array_for_each_unique(&hashes, print_callback, NULL);
}
```

Last updated 2014-11-27 19:55:05 CET

sigchain API

Code often wants to set a signal handler to clean up temporary files or other work-in-progress when we die unexpectedly. For multiple pieces of code to do this without conflicting, each piece of code must remember the old value of the handler and restore it either when:

1. The work-in-progress is finished, and the handler is no longer necessary. The handler should revert to the original behavior (either another handler, SIG_DFL, or SIG_IGN).
2. The signal is received. We should then do our cleanup, then chain to the next handler (or die if it is SIG_DFL).

Sigchain is a tiny library for keeping a stack of handlers. Your handler and installation code should look something like:

```
void clean_foo_on_signal(int sig)
{
        clean_foo();
        sigchain_pop(sig);
        raise(sig);
}

void other_func()
{
        sigchain_push_common(clean_foo_on_signal);
        mess_up_foo();
        clean_foo();
}
```

Handlers are given the typedef of sigchain_fun. This is the same type that is given to signal() or sigaction(). It is perfectly reasonable to push SIG_DFL or SIG_IGN onto the stack.

You can sigchain_push and sigchain_pop individual signals. For convenience, sigchain_push_common will push the handler onto the stack for many common signals.

Last updated 2014-01-25 09:03:55 CET

string-list API

The string_list API offers a data structure and functions to handle sorted and unsorted string lists. A "sorted" list is one whose entries are sorted by string value in `strcmp()` order.

The *string_list* struct used to be called *path_list*, but was renamed because it is not specific to paths.

The caller:

1. Allocates and clears a `struct string_list` variable.
2. Initializes the members. You might want to set the flag `strdup_strings` if the strings should be strdup()ed. For example, this is necessary when you add something like git_path("..."), since that function returns a static buffer that will change with the next call to git_path().

   If you need something advanced, you can manually malloc() the `items` member (you need this if you add things later) and you should set the `nr` and `alloc` members in that case, too.
3. Adds new items to the list, using `string_list_append`, `string_list_append_nodup`, `string_list_insert`, `string_list_split`, and/or `string_list_split_in_place`.
4. Can check if a string is in the list using `string_list_has_string` or `unsorted_string_list_has_string` and get it from the list using `string_list_lookup` for sorted lists.
5. Can sort an unsorted list using `string_list_sort`.
6. Can remove duplicate items from a sorted list using `string_list_remove_duplicates`.
7. Can remove individual items of an unsorted list using `unsorted_string_list_delete_item`.
8. Can remove items not matching a criterion from a sorted or unsorted list using `filter_string_list`, or remove empty strings using `string_list_remove_empty_items`.
9. Finally it should free the list using `string_list_clear`.

Example:

```
struct string_list list = STRING_LIST_INIT_NODUP;
int i;

string_list_append(&list, "foo");
string_list_append(&list, "bar");
for (i = 0; i < list.nr; i++)
        printf("%s\n", list.items[i].string)
```

Note It is more efficient to build an unsorted list and sort it afterwards, instead of building a sorted list (`O(n log n)` instead of `O(n^2)`).

+ However, if you use the list to check if a certain string was added already, you should not do that (using unsorted_string_list_has_string()), because the complexity would be quadratic again (but with a worse factor).

Functions

- General ones (works with sorted and unsorted lists as well)

  `string_list_init`
  : Initialize the members of the string_list, set `strdup_strings` member according to the value of the second parameter.

  `filter_string_list`
  : Apply a function to each item in a list, retaining only the items for which the function returns true. If free_util is true, call free() on the util members of any items that have to be deleted. Preserve the order of the items that are retained.

  `string_list_remove_empty_items`
  : Remove any empty strings from the list. If free_util is true, call free() on the util members of any items that have to be deleted. Preserve the order of the items that are retained.

  `print_string_list`
  : Dump a string_list to stdout, useful mainly for debugging purposes. It can take an optional header argument and it writes out the string-pointer pairs of the string_list, each one in its own line.

  `string_list_clear`
  : Free a string_list. The `string` pointer of the items will be freed in case the `strdup_strings` member of the string_list is set. The second parameter controls if the `util` pointer of the items should be freed or not.

- Functions for sorted lists only

  `string_list_has_string`
  : Determine if the string_list has a given string or not.

  `string_list_insert`
  : Insert a new element to the string_list. The returned pointer can be handy if you want to write something to the `util` pointer of the string_list_item containing the just added string. If the given string already exists the insertion will be skipped and the pointer to the existing item returned.

    Since this function uses xrealloc() (which die()s if it fails) if the list needs to grow, it is safe not to check the pointer. I.e. you may write `string_list_insert(...)->util = ...;`.

  `string_list_lookup`
  : Look up a given string in the string_list, returning the containing string_list_item. If the string is not found, NULL is returned.

  `string_list_remove_duplicates`
  : Remove all but the first of consecutive entries that have the same string value. If free_util is true, call free() on the util members of any items that have to be deleted.

- Functions for unsorted lists only

  `string_list_append`
  : Append a new string to the end of the string_list. If `strdup_string` is set, then the string argument is copied; otherwise the new `string_list_entry` refers to the input string.

  `string_list_append_nodup`
  : Append a new string to the end of the string_list. The new `string_list_entry` always refers to the input string, even if `strdup_string` is set. This function can be used to hand ownership of a malloc()ed string to a `string_list` that has `strdup_string` set.

  `string_list_sort`
  : Sort the list's entries by string value in `strcmp()` order.

  `unsorted_string_list_has_string`
  : It's like `string_list_has_string()` but for unsorted lists.

  `unsorted_string_list_lookup`
  : It's like `string_list_lookup()` but for unsorted lists.

    The above two functions need to look through all items, as opposed to their counterpart for sorted lists, which performs a binary search.

  `unsorted_string_list_delete_item`
  : Remove an item from a string_list. The `string` pointer of the items will be freed in case the `strdup_strings` member of the string_list is set. The third parameter controls if the `util` pointer of the items should be freed or not.

  `string_list_split`

  `string_list_split_in_place`
  : Split a string into substrings on a delimiter character and append the substrings to a `string_list`. If `maxsplit` is non-negative, then split at most `maxsplit` times. Return the number of substrings appended to

the list.

`string_list_split` requires a `string_list` that has `strdup_strings` set to true; it leaves the input string untouched and makes copies of the substrings in newly-allocated memory. `string_list_split_in_place` requires a `string_list` that has `strdup_strings` set to false; it splits the input string in place, overwriting the delimiter characters with NULs and creating new string_list_items that point into the original string (the original string must therefore not be modified or freed while the `string_list` is in use).

Data structures

- `struct string_list_item`

Represents an item of the list. The `string` member is a pointer to the string, and you may use the `util` member for any purpose, if you want.

- `struct string_list`

Represents the list itself.

1. The array of items are available via the `items` member.
2. The `nr` member contains the number of items stored in the list.
3. The `alloc` member is used to avoid reallocating at every insertion. You should not tamper with it.
4. Setting the `strdup_strings` member to 1 will strdup() the strings before adding them, see above.
5. The `compare_strings_fn` member is used to specify a custom compare function, otherwise `strcmp()` is used as the default function.

Last updated 2014-12-23 17:41:32 CET

trace API

The trace API can be used to print debug messages to stderr or a file. Trace code is inactive unless explicitly enabled by setting `GIT_TRACE*` environment variables.

The trace implementation automatically adds `timestamp file:line ... \n` to all trace messages. E.g.:

```
23:59:59.123456 git.c:312               trace: built-in: git 'foo'
00:00:00.000001 builtin/foo.c:99        foo: some message
```

Data Structures

`struct trace_key`

Defines a trace key (or category). The default (for API functions that don't take a key) is `GIT_TRACE`.

E.g. to define a trace key controlled by environment variable `GIT_TRACE_FOO`:

```
static struct trace_key trace_foo = TRACE_KEY_INIT(FOO);

static void trace_print_foo(const char *message)
{
        trace_print_key(&trace_foo, message);
}
```

Note: don't use `const` as the trace implementation stores internal state in the `trace_key` structure.

Functions

`int trace_want(struct trace_key *key)`

Checks whether the trace key is enabled. Used to prevent expensive string formatting before calling one of the printing APIs.

`void trace_disable(struct trace_key *key)`

Disables tracing for the specified key, even if the environment variable was set.

`void trace_printf(const char *format, ...)`

`void trace_printf_key(struct trace_key *key, const char *format, ...)`
: Prints a formatted message, similar to printf.

`void trace_argv_printf(const char **argv, const char *format, ...)``
: Prints a formatted message, followed by a quoted list of arguments.

`void trace_strbuf(struct trace_key *key, const struct strbuf *data)`
: Prints the strbuf, without additional formatting (i.e. doesn't choke on `%` or even `\0`).

`uint64_t getnanotime(void)`
: Returns nanoseconds since the epoch (01/01/1970), typically used for performance measurements.

  Currently there are high precision timer implementations for Linux (using `clock_gettime(CLOCK_MONOTONIC)`) and Windows (`QueryPerformanceCounter`). Other platforms use `gettimeofday` as time source.

`void trace_performance(uint64_t nanos, const char *format, ...)`

`void trace_performance_since(uint64_t start, const char *format, ...)`
: Prints the elapsed time (in nanoseconds), or elapsed time since `start`, followed by a formatted message. Enabled via environment variable `GIT_TRACE_PERFORMANCE`. Used for manual profiling, e.g.:

```
uint64_t start = getnanotime();
/* code section to measure */
trace_performance_since(start, "foobar");
```

```
uint64_t t = 0;
for (;;) {
        /* ignore */
        t -= getnanotime();
        /* code section to measure */
        t += getnanotime();
        /* ignore */
}
trace_performance(t, "frotz");
```

Last updated 2014-11-27 19:58:08 CET

tree walking API

The tree walking API is used to traverse and inspect trees.

Data Structures

`struct name_entry`
: An entry in a tree. Each entry has a sha1 identifier, pathname, and mode.

`struct tree_desc`
: A semi-opaque data structure used to maintain the current state of the walk.

  - `buffer` is a pointer into the memory representation of the tree. It always points at the current entry being visited.
  - `size` counts the number of bytes left in the `buffer`.
  - `entry` points to the current entry being visited.

`struct traverse_info`
: A structure used to maintain the state of a traversal.

  - `prev` points to the traverse_info which was used to descend into the current tree. If this is the top-level tree `prev` will point to a dummy traverse_info.
  - `name` is the entry for the current tree (if the tree is a subtree).
  - `pathlen` is the length of the full path for the current tree.
  - `conflicts` can be used by callbacks to maintain directory-file conflicts.
  - `fn` is a callback called for each entry in the tree. See Traversing for more information.
  - `data` can be anything the `fn` callback would want to use.

- `show_all_errors` tells whether to stop at the first error or not.

Initializing

`init_tree_desc`
: Initialize a `tree_desc` and decode its first entry. The buffer and size parameters are assumed to be the same as the buffer and size members of `struct tree`.

`fill_tree_descriptor`
: Initialize a `tree_desc` and decode its first entry given the sha1 of a tree. Returns the `buffer` member if the sha1 is a valid tree identifier and NULL otherwise.

`setup_traverse_info`
: Initialize a `traverse_info` given the pathname of the tree to start traversing from. The `base` argument is assumed to be the `path` member of the `name_entry` being recursed into unless the tree is a top-level tree in which case the empty string ("") is used.

Walking

`tree_entry`
: Visit the next entry in a tree. Returns 1 when there are more entries left to visit and 0 when all entries have been visited. This is commonly used in the test of a while loop.

`tree_entry_len`
: Calculate the length of a tree entry's pathname. This utilizes the memory structure of a tree entry to avoid the overhead of using a generic strlen().

`update_tree_entry`
: Walk to the next entry in a tree. This is commonly used in conjunction with `tree_entry_extract` to inspect the current entry.

`tree_entry_extract`
: Decode the entry currently being visited (the one pointed to by `tree_desc's entry` member) and return the sha1 of the entry. The `pathp` and `modep` arguments are set to the entry's pathname and mode respectively.

`get_tree_entry`
: Find an entry in a tree given a pathname and the sha1 of a tree to search. Returns 0 if the entry is found and -1 otherwise. The third and fourth parameters are set to the entry's sha1 and mode respectively.

Traversing

`traverse_trees`
: Traverse `n` number of trees in parallel. The `fn` callback member of `traverse_info` is called once for each tree entry.

`traverse_callback_t`
: The arguments passed to the traverse callback are as follows:

  - `n` counts the number of trees being traversed.
  - `mask` has its nth bit set if something exists in the nth entry.
  - `dirmask` has its nth bit set if the nth tree's entry is a directory.
  - `entry` is an array of size `n` where the nth entry is from the nth tree.
  - `info` maintains the state of the traversal.

  Returning a negative value will terminate the traversal. Otherwise the return value is treated as an update mask. If the nth bit is set the nth tree will be updated and if the bit is not set the nth tree entry will be the same in the next callback invocation.

`make_traverse_path`
: Generate the full pathname of a tree entry based from the root of the traversal. For example, if the traversal has recursed into another tree named "bar" the pathname of an entry "baz" in the "bar" tree would be "bar/baz".

`traverse_path_len`
: Calculate the length of a pathname returned by `make_traverse_path`. This utilizes the memory structure of a tree entry to avoid the overhead of using a generic strlen().

Authors

Written by Junio C Hamano <gitster@pobox.com> and Linus Torvalds <torvalds@linux-foundation.org>

Last updated 2014-01-25 09:03:55 CET

xdiff interface API

Talk about our calling convention to xdiff library, including xdiff_emit_consume_fn.
(Dscho, JC)

Last updated 2014-01-25 09:03:55 CET

GIT bitmap v1 format

- A header appears at the beginning:

```
4-byte signature: {'B', 'I', 'T', 'M'}

2-byte version number (network byte order)
        The current implementation only supports version 1
        of the bitmap index (the same one as JGit).

2-byte flags (network byte order)

The following flags are supported:
```

- BITMAP_OPT_FULL_DAG (0x1) REQUIRED This flag must always be present. It implies that the bitmap index has been generated for a packfile with full closure (i.e. where every single object in the packfile can find its parent links inside the same packfile). This is a requirement for the bitmap index format, also present in JGit, that greatly reduces the complexity of the implementation.
- BITMAP_OPT_HASH_CACHE (0x4) If present, the end of the bitmap file contains `N` 32-bit name-hash values, one per object in the pack. The format and meaning of the name-hash is described below.

```
4-byte entry count (network byte order)

The total count of entries (bitmapped commits) in this bitmap index.

20-byte checksum

The SHA1 checksum of the pack this bitmap index belongs to.
```

- 4 EWAH bitmaps that act as type indexes

```
Type indexes are serialized after the hash cache in the shape
of four EWAH bitmaps stored consecutively (see Appendix A for
the serialization format of an EWAH bitmap).

There is a bitmap for each Git object type, stored in the following
order:
```

- Commits
- Trees
- Blobs
- Tags

```
In each bitmap, the `n`th bit is set to true if the `n`th object
in the packfile is of that type.

The obvious consequence is that the OR of all 4 bitmaps will result
in a full set (all bits set), and the AND of all 4 bitmaps will
result in an empty bitmap (no bits set).
```

- N entries with compressed bitmaps, one for each indexed commit

```
Where `N` is the total amount of entries in this bitmap index.
Each entry contains the following:
```

- 4-byte object position (network byte order) The position **in the index for the packfile** where the bitmap for this commit is found.
- 1-byte XOR-offset The xor offset used to compress this bitmap. For an entry in position `x`, a XOR offset of `y` means that the actual bitmap representing this commit is composed by XORing the bitmap for this entry with the bitmap in entry `x-y` (i.e. the bitmap `y` entries before this one).

```
Note that this compression can be recursive. In order to
XOR this entry with a previous one, the previous entry needs
to be decompressed first, and so on.

The hard-limit for this offset is 160 (an entry can only be
xor'ed against one of the 160 entries preceding it). This
number is always positive, and hence entries are always xor'ed
with **previous** bitmaps, not bitmaps that will come afterwards
in the index.
```

- 1-byte flags for this bitmap At the moment the only available flag is `0x1`, which hints that this bitmap can be re-used when rebuilding bitmap indexes for the repository.
- The compressed bitmap itself, see Appendix A.

Appendix A: Serialization format for an EWAH bitmap

Ewah bitmaps are serialized in the same protocol as the JAVAEWAH library, making them backwards compatible with the JGit implementation:

- 4-byte number of bits of the resulting UNCOMPRESSED bitmap
- 4-byte number of words of the COMPRESSED bitmap, when stored
- N x 8-byte words, as specified by the previous field

  ```
  This is the actual content of the compressed bitmap.
  ```

- 4-byte position of the current RLW for the compressed bitmap

All words are stored in network byte order for their corresponding sizes.

The compressed bitmap is stored in a form of run-length encoding, as follows. It consists of a concatenation of an arbitrary number of chunks. Each chunk consists of one or more 64-bit words

```
H  L_1  L_2  L_3 .... L_M
```

H is called RLW (run length word). It consists of (from lower to higher order bits):

- 1 bit: the repeated bit B
- 32 bits: repetition count K (unsigned)
- 31 bits: literal word count M (unsigned)

The bitstream represented by the above chunk is then:

- K repetitions of B
- The bits stored in `L_1` through `L_M`. Within a word, bits at lower order come earlier in the stream than those at higher order.

The next word after `L_M` (if any) must again be a RLW, for the next chunk. For efficient appending to the bitstream, the EWAH stores a pointer to the last RLW in the stream.

Appendix B: Optional Bitmap Sections

These sections may or may not be present in the `.bitmap` file; their presence is indicated by the header flags section described above.

Name-hash cache

If the BITMAP_OPT_HASH_CACHE flag is set, the end of the bitmap contains a cache of 32-bit values, one per object in the pack. The value at position `i` is the hash of the pathname at which the `i`th object (counting in index order) in the pack can be found. This can be fed into the delta heuristics to compare objects with similar pathnames.

The hash algorithm used is:

```
hash = 0;
while ((c = *name++))
        if (!isspace(c))
                hash = (hash >> 2) + (c << 24);
```

Note that this hashing scheme is tied to the BITMAP_OPT_HASH_CACHE flag. If implementations want to choose a different hashing scheme, they are free to do so, but MUST allocate a new header flag (because comparing hashes made under two different schemes would be pointless).

Last updated 2014-11-27 19:57:04 CET

Coding guidelines

Like other projects, we also have some guidelines to keep to the code. For Git in general, a few rough rules are:

- Most importantly, we never say "It's in POSIX; we'll happily ignore your needs should your system not conform to it." We live in the real world.
- However, we often say "Let's stay away from that construct, it's not even in POSIX".
- In spite of the above two rules, we sometimes say "Although this is not in POSIX, it (is so convenient | makes the code much more readable | has other good characteristics) and practically all the platforms we care about support it, so let's use it".
  Again, we live in the real world, and it is sometimes a judgement call, the decision based more on real world constraints people face than what the paper standard says.
- Fixing style violations while working on a real change as a preparatory clean-up step is good, but otherwise avoid useless code churn for the sake of conforming to the style.
  "Once it *is* in the tree, it's not really worth the patch noise to go and fix it up." Cf. http://article.gmane.org/gmane.linux.kernel/943020

Make your code readable and sensible, and don't try to be clever.

As for more concrete guidelines, just imitate the existing code (this is a good guideline, no matter which project you are contributing to). It is always preferable to match the *local* convention. New code added to Git suite is expected to match the overall style of existing code. Modifications to existing code is expected to match the style the surrounding code already uses (even if it doesn't match the overall style of existing code).

But if you must have a list of rules, here they are.

For shell scripts specifically (not exhaustive):

- We use tabs for indentation.
- Case arms are indented at the same depth as case and esac lines, like this:

```
case "$variable" in
pattern1)
        do this
        ;;
pattern2)
        do that
        ;;
esac
```

- Redirection operators should be written with space before, but no space after them. In other words, write *echo test >"$file"* instead of *echo test> $file* or *echo test > $file*. Note that even though it is not required by POSIX to double-quote the redirection target in a variable (as shown above), our code does so because some versions of bash issue a warning without the quotes.

```
(incorrect)
cat hello > world < universe
echo hello >$world

(correct)
cat hello >world <universe
echo hello >"$world"
```

- We prefer $(...) for command substitution; unlike ``, it properly nests. It should have been the way Bourne spelled it from day one, but unfortunately isn't.
- If you want to find out if a command is available on the user's $PATH, you should use *type <command>*, instead of *which <command>*. The output of *which* is not machine parseable and its exit code is not reliable across platforms.
- We use POSIX compliant parameter substitutions and avoid bashisms; namely:
  - We use `${parameter-word}` and its `[-=?+]` siblings, and their colon'ed "unset or null" form.
  - We use `${parameter#word}` and its `[#%]` siblings, and their doubled "longest matching" form.
  - No "Substring Expansion" `${parameter:offset:length}`.
  - No shell arrays.
  - No strlen `${#parameter}`.
  - No pattern replacement `${parameter/pattern/string}`.
- We use Arithmetic Expansion $((...)).
- Inside Arithmetic Expansion, spell shell variables with $ in front of them, as some shells do not grok $x)) while accepting $(($x just fine (e.g. dash older than 0.5.4).
- We do not use Process Substitution <(list) or >(list).

- Do not write control structures on a single line with semicolon. "then" should be on the next line for if statements, and "do" should be on the next line for "while" and "for".

  ```
  (incorrect)
  if test -f hello; then
          do this
  fi

  (correct)
  if test -f hello
  then
          do this
  fi
  ```

- We prefer "test" over "[...]".
- We do not write the noiseword "function" in front of shell functions.
- We prefer a space between the function name and the parentheses, and no space inside the parentheses. The opening "{" should also be on the same line.

  ```
  (incorrect)
  my_function(){
          ...

  (correct)
  my_function () {
          ...
  ```

- As to use of grep, stick to a subset of BRE (namely, no {m,n}, [::], [==], or [..]) for portability.
  - We do not use {m,n};
  - We do not use -E;
  - We do not use ? or + (which are {0,1} and {1,} respectively in BRE) but that goes without saying as these are ERE elements not BRE (note that \? and \+ are not even part of BRE — making them accessible from BRE is a GNU extension).
- Use Git’s gettext wrappers in git-sh-i18n to make the user interface translatable. See "Marking strings for translation" in po/README.
- We do not write our "test" command with "-a" and "-o" and use "&&" or "||" to concatenate multiple "test" commands instead, because the use of "-a/-o" is often error-prone. E.g.

  ```
  test -n "$x" -a "$a" = "$b"
  ```

  is buggy and breaks when $x is "=", but

  ```
  test -n "$x" && test "$a" = "$b"
  ```

  does not have such a problem.

For C programs:

- We use tabs to indent, and interpret tabs as taking up to 8 spaces.
- We try to keep to at most 80 characters per line.
- We try to support a wide range of C compilers to compile Git with, including old ones. That means that you should not use C99 initializers, even if a lot of compilers grok it.
- Variables have to be declared at the beginning of the block.
- NULL pointers shall be written as NULL, not as 0.
- When declaring pointers, the star sides with the variable name, i.e. `char *string`, not `char* string` or `char * string`. This makes it easier to understand code like `char *string, c;`.
- Use whitespace around operators and keywords, but not inside parentheses and not around functions. So:

  ```
  while (condition)
  func(bar + 1);
  ```

  and not:

  ```
  while( condition )
  func (bar+1);
  ```

- We avoid using braces unnecessarily. I.e.

  ```
  if (bla) {
          x = 1;
  }
  ```

is frowned upon. A gray area is when the statement extends over a few lines, and/or you have a lengthy comment atop of it. Also, like in the Linux kernel, if there is a long list of `else if` statements, it can make sense to add braces to single line blocks.

- We try to avoid assignments inside if().
- Try to make your code understandable. You may put comments in, but comments invariably tend to stale out when the code they were describing changes. Often splitting a function into two makes the intention of the code much clearer.
- Multi-line comments include their delimiters on separate lines from the text. E.g.

  ```
  /*
   * A very long
   * multi-line comment.
   */
  ```

  Note however that a comment that explains a translatable string to translators uses a convention of starting with a magic token "TRANSLATORS: " immediately after the opening delimiter, even when it spans multiple lines. We do not add an asterisk at the beginning of each line, either. E.g.

  ```
  /* TRANSLATORS: here is a comment that explains the string
     to be translated, that follows immediately after it */
  _("Here is a translatable string explained by the above.");
  ```

- Double negation is often harder to understand than no negation at all.
- There are two schools of thought when it comes to comparison, especially inside a loop. Some people prefer to have the less stable value on the left hand side and the more stable value on the right hand side, e.g. if you have a loop that counts variable i down to the lower bound,

  ```
  while (i > lower_bound) {
    do something;
      i--;
  }
  ```

  Other people prefer to have the textual order of values match the actual order of values in their comparison, so that they can mentally draw a number line from left to right and place these values in order, i.e.

  ```
  while (lower_bound < i) {
    do something;
    i--;
  }
  ```

  Both are valid, and we use both. However, the more "stable" the stable side becomes, the more we tend to prefer the former (comparison with a constant, "i > 0", is an extreme example). Just do not mix styles in the same part of the code and mimic existing styles in the neighbourhood.

- There are two schools of thought when it comes to splitting a long logical line into multiple lines. Some people push the second and subsequent lines far enough to the right with tabs and align them:

  ```
  if (the_beginning_of_a_very_long_expression_that_has_to ||
          span_more_than_a_single_line_of ||
          the_source_text) {
          ...
  ```

  while other people prefer to align the second and the subsequent lines with the column immediately inside the opening parenthesis, with tabs and spaces, following our "tabstop is always a multiple of 8" convention:

  ```
  if (the_beginning_of_a_very_long_expression_that_has_to ||
      span_more_than_a_single_line_of ||
      the_source_text) {
          ...
  ```

  Both are valid, and we use both. Again, just do not mix styles in the same part of the code and mimic existing styles in the neighbourhood.

- When splitting a long logical line, some people change line before a binary operator, so that the result looks like a parse tree when you turn your head 90-degrees counterclockwise:

  ```
  if (the_beginning_of_a_very_long_expression_that_has_to
      || span_more_than_a_single_line_of_the_source_text) {
  ```

  while other people prefer to leave the operator at the end of the line:

  ```
  if (the_beginning_of_a_very_long_expression_that_has_to ||
      span_more_than_a_single_line_of_the_source_text) {
  ```

  Both are valid, but we tend to use the latter more, unless the expression gets fairly complex, in which case the former tends to be easier to read. Again, just do not mix styles in the same part of the code and mimic existing styles in the neighbourhood.

- When splitting a long logical line, with everything else being equal, it is preferable to split after the operator at higher level in the parse tree. That is, this is more preferable:

  ```
  if (a_very_long_variable * that_is_used_in +
      a_very_long_expression) {
          ...
  ```

  than

  ```
  if (a_very_long_variable *
      that_is_used_in + a_very_long_expression) {
          ...
  ```

- Some clever tricks, like using the !! operator with arithmetic constructs, can be extremely confusing to others. Avoid them, unless there is a compelling reason to use them.
- Use the API. No, really. We have a strbuf (variable length string), several arrays with the ALLOC_GROW() macro, a string_list for sorted string lists, a hash map (mapping struct objects) named "struct decorate", amongst other things.
- When you come up with an API, document it.
- The first #include in C files, except in platform specific compat/ implementations, must be either "git-compat-util.h", "cache.h" or "builtin.h". You do not have to include more than one of these.
- A C file must directly include the header files that declare the functions and the types it uses, except for the functions and types that are made available to it by including one of the header files it must include by the previous rule.
- If you are planning a new command, consider writing it in shell or perl first, so that changes in semantics can be easily changed and discussed. Many Git commands started out like that, and a few are still scripts.
- Avoid introducing a new dependency into Git. This means you usually should stay away from scripting languages not already used in the Git core command set (unless your command is clearly separate from it, such as an importer to convert random-scm-X repositories to Git).
- When we pass <string, length> pair to functions, we should try to pass them in that order.
- Use Git's gettext wrappers to make the user interface translatable. See "Marking strings for translation" in po/README.

For Perl programs:

- Most of the C guidelines above apply.
- We try to support Perl 5.8 and later ("use Perl 5.008").
- use strict and use warnings are strongly preferred.
- Don't overuse statement modifiers unless using them makes the result easier to follow.

  ```
          ... do something ...
          do_this() unless (condition);
          ... do something else ...
  ```

  is more readable than:

  ```
          ... do something ...
          unless (condition) {
                  do_this();
          }
          ... do something else ...
  ```

only when the condition is so rare that do_this() will be almost always called.

- We try to avoid assignments inside "if ()" conditions.
- Learn and use Git.pm if you need that functionality.
- For Emacs, it's useful to put the following in GIT_CHECKOUT/.dir-locals.el, assuming you use cperl-mode:

  > note the first part is useful for C editing, too ((nil . ((indent-tabs-mode . t) (tab-width . 8) (fill-column . 80))) (cperl-mode . ((cperl-indent-level . 8) (cperl-extra-newline-before-brace . nil) (cperl-merge-trailing-else . t))))

For Python scripts:

- We follow PEP-8 (http://www.python.org/dev/peps/pep-0008/).
- As a minimum, we aim to be compatible with Python 2.6 and 2.7.
- Where required libraries do not restrict us to Python 2, we try to also be compatible with Python 3.1 and later.
- When you must differentiate between Unicode literals and byte string literals, it is OK to use the *b* prefix. Even though the Python documentation for version 2.6 does not mention this prefix, it has been supported since version 2.6.0.

Error Messages

- Do not end error messages with a full stop.
- Do not capitalize ("unable to open %s", not "Unable to open %s")
- Say what the error is first ("cannot open %s", not "%s: cannot open")

Externally Visible Names

- For configuration variable names, follow the existing convention:
    1. The section name indicates the affected subsystem.
    2. The subsection name, if any, indicates which of an unbounded set of things to set the value for.
    3. The variable name describes the effect of tweaking this knob.

```
The section and variable names that consist of multiple words are
formed by concatenating the words without punctuations (e.g. `-`),
and are broken using bumpyCaps in documentation as a hint to the
reader.

When choosing the variable namespace, do not use variable name for
specifying possibly unbounded set of things, most notably anything
an end user can freely come up with (e.g. branch names).  Instead,
use subsection names or variable values, like the existing variable
branch.<name>.description does.
```

Writing Documentation:

Most (if not all) of the documentation pages are written in the AsciiDoc format in *.txt files (e.g. Documentation/git.txt), and processed into HTML and manpages (e.g. git.html and git.1 in the same directory).

The documentation liberally mixes US and UK English (en_US/UK) norms for spelling and grammar, which is somewhat unfortunate. In an ideal world, it would have been better if it consistently used only one and not the other, and we would have picked en_US (if you wish to correct the English of some of the existing documentation, please see the documentation-related advice in the Documentation/SubmittingPatches file).

Every user-visible change should be reflected in the documentation. The same general rule as for code applies — imitate the existing conventions.

A few commented examples follow to provide reference when writing or modifying command usage strings and synopsis sections in the manual pages:

Placeholders are spelled in lowercase and enclosed in angle brackets:

```
<file>
--sort=<key>
--abbrev[=<n>]
```

If a placeholder has multiple words, they are separated by dashes: <new-branch-name> --template=<template-directory>

Possibility of multiple occurrences is indicated by three dots:

```
<file>...
(One or more of <file>.)
```

Optional parts are enclosed in square brackets:

```
[<extra>]
(Zero or one <extra>.)

--exec-path[=<path>]
(Option with an optional argument.  Note that the "=" is inside the
brackets.)

[<patch>...]
(Zero or more of <patch>.  Note that the dots are inside, not
outside the brackets.)
```

Multiple alternatives are indicated with vertical bars:

```
[-q | --quiet]
[--utf8 | --no-utf8]
```

Parentheses are used for grouping:

```
[(<rev> | <range>)...]
(Any number of either <rev> or <range>.  Parens are needed to make
it clear that "..." pertains to both <rev> and <range>.)

[(-p <parent>)...]
(Any number of option -p, each with one <parent> argument.)
```

```
   git remote set-head <name> (-a | -d | <branch>)
   (One and only one of "-a", "-d" or "<branch>" _must_ (no square
   brackets) be provided.)
```

And a somewhat more contrived example:

```
   --diff-filter=[(A|C|D|M|R|T|U|X|B)...[*]]
   Here "=" is outside the brackets, because "--diff-filter=" is a
   valid usage.  "*" has its own pair of brackets, because it can
   (optionally) be specified only when one or more of the letters is
   also provided.
```

A note on notation: Use *git* (all lowercase) when talking about commands i.e. something the user would type into a shell and use *Git* (uppercase first letter) when talking about the version control system and its properties.

A few commented examples follow to provide reference when writing or modifying paragraphs or option/command explanations that contain options or commands:

Literal examples (e.g. use of command-line options, command names, and configuration variables) are typeset in monospace, and if you can use `backticks around word phrases`, do so.

```
`--pretty=oneline`
`git rev-list`
`remote.pushDefault`
```

Word phrases enclosed in `backtick characters` are rendered literally and will not be further expanded. The use of `backticks` to achieve the previous rule means that literal examples should not use AsciiDoc escapes.

```
Correct:
   `--pretty=oneline`
Incorrect:
   `\--pretty=oneline`
```

If some place in the documentation needs to typeset a command usage example with inline substitutions, it is fine to use `monospaced and inline substituted text` instead of `monospaced literal text`, and with the former, the part that should not get substituted must be quoted/escaped.

Last updated 2015-05-03 21:16:55 CEST

HTTP transfer protocols

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 for 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.

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
```

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: 004895dcfa3633004da0049d3d0fa03f80589cbcaf31 refs/heads/maint\0multi_ack\n
S: 0042d049f6c27a2244e12041955e262a404c7faba355 refs/heads/master\n
S: 003c2cb58b79488a98d2721cea644875a8dd0026b115 refs/tags/v1.0\n
S: 003fa3c2e2402b99163d1d59756e5f207ae21cccba4c refs/tags/v1.0^{}\n
```

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.

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.

```
smart_reply     =  PKT-LINE("# service=$servicename" LF)
                   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`.

```
compute_request   =  want_list
                     have_list
                     request_end
request_end       =  "0000" / "done"

want_list         =  PKT-LINE(want NUL cap_list LF)
                     *(want_pkt)
want_pkt          =  PKT-LINE(want LF)
want              =  "want" SP id
cap_list          =  *(SP capability) SP
```

```
have_list         =  *PKT-LINE("have" SP id LF)
```

TODO: Document this further.

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` 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 SHA-1 as its value. Multiple SHA-1s MUST be sent by sending multiple commands.

The `have` list is created by popping the first 32 commits from `c_pending`. Less 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 r
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) RFC 2616: Hypertext Transfer Protocol — HTTP/1.1
link:technical/pack-protocol.html link:technical/protocol-capabilities.html

Last updated 2014-11-27 19:58:08 CET

Git index format

The Git index file has the following format

```
All binary numbers are in network byte order. 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.

Git currently supports cached tree and resolve undo extensions.

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
```

- 160-bit SHA-1 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.

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)

4-bit object type
  valid values in binary are 1000 (regular file), 1010 (symbolic link)
  and 1110 (gitlink)

3-bit unused

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.

160-bit SHA-1 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 pack-format.txt) 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

Cached tree

```
Cached tree extension contains pre-computed hashes for trees that can
be derived from the index. It helps speed up tree object generation
from index for a new commit.

When a path is updated in index, the path must be invalidated and
removed from tree cache.

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
- 160-bit 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), ...
```

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 160-bit object names of the entry in stages from 1 to 3 (nothing is written for a missing stage).

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:
```

- 160-bit SHA-1 of the shared index file. The shared index file path is $GIT_DIR/sharedindex.<SHA-1>. If all 160 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.
  ```

Last updated 2014-12-23 17:41:32 CET

Git pack format

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 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)
20-byte 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: length of each object is encoded in a variable
length format and is not constrained to 32-bit or anything.
```

- The trailer records 20-byte SHA-1 checksum of all of the above.

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.

20-byte object name.
```

- The file is concluded with a trailer:

```
A copy of the 20-byte SHA-1 checksum at the end of
corresponding packfile.

20-byte SHA-1-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
     20-byte base object name SHA-1 (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 20-byte SHA-1 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 20-byte SHA-1 checksum at the end of
  corresponding packfile.

  20-byte SHA-1-checksum of all of the above.
  ```

Last updated 2014-11-27 19:55:05 CET

Concerning Git’s Packing Heuristics

```
Oh, here's a really stupid question:

      Where do I go
   to learn the details
of Git's packing heuristics?
```

Be careful what you ask!

Followers of the Git, please open the Git IRC Log and turn to February 10, 2006.

It’s a rare occasion, and we are joined by the King Git Himself, Linus Torvalds (linus). Nathaniel Smith, (njs`), has the floor and seeks enlightenment. Others are present, but silent.

Let’s listen in!

```
<njs`> Oh, here's a really stupid question -- where do I go to
```

```
    learn the details of Git's packing heuristics?  google avails
    me not, reading the source didn't help a lot, and wading
    through the whole mailing list seems less efficient than any
    of that.
```

It is a bold start! A plea for help combined with a simultaneous tri-part attack on some of the tried and true mainstays in the quest for enlightenment. Brash accusations of google being useless. Hubris! Maligning the source. Heresy! Disdain for the mailing list archives. Woe.

```
<pasky> yes, the packing-related delta stuff is somewhat
    mysterious even for me ;)
```

Ah! Modesty after all.

```
<linus> njs, I don't think the docs exist. That's something where
     I don't think anybody else than me even really got involved.
     Most of the rest of Git others have been busy with (especially
     Junio), but packing nobody touched after I did it.
```

It’s cryptic, yet vague. Linus in style for sure. Wise men interpret this as an apology. A few argue it is merely a statement of fact.

```
<njs`> I guess the next step is "read the source again", but I
    have to build up a certain level of gumption first :-)
```

Indeed! On both points.

```
<linus> The packing heuristic is actually really really simple.
```

Bait...

```
<linus> But strange.
```

And switch. That ought to do it!

```
<linus> Remember: Git really doesn't follow files. So what it does is
    - generate a list of all objects
    - sort the list according to magic heuristics
    - walk the list, using a sliding window, seeing if an object
      can be diffed against another object in the window
    - write out the list in recency order
```

The traditional understatement:

```
<njs`> I suspect that what I'm missing is the precise definition of
    the word "magic"
```

The traditional insight:

```
<pasky> yes
```

And Babel-like confusion flowed.

```
<njs`> oh, hmm, and I'm not sure what this sliding window means either
```

```
<pasky> iirc, it appeared to me to be just the sha1 of the object
    when reading the code casually ...
```

i. which simply doesn’t sound as a very good heuristics, though ;)

```
<njs`> .....and recency order.  okay, I think it's clear I didn't
   even realize how much I wasn't realizing :-)
```

Ah, grasshopper! And thus the enlightenment begins anew.

```
<linus> The "magic" is actually in theory totally arbitrary.
    ANY order will give you a working pack, but no, it's not
    ordered by SHA-1.
```

```
Before talking about the ordering for the sliding delta
window, let's talk about the recency order. That's more
important in one way.
```

```
<njs`> Right, but if all you want is a working way to pack things
    together, you could just use cat and save yourself some
    trouble...
```

Waaait for it....

```
<linus> The recency ordering (which is basically: put objects
    _physically_ into the pack in the order that they are
    "reachable" from the head) is important.


<njs`> okay


<linus> It's important because that's the thing that gives packs
    good locality. It keeps the objects close to the head (whether
    they are old or new, but they are _reachable_ from the head)
    at the head of the pack. So packs actually have absolutely
    _wonderful_ IO patterns.
```

Read that again, because it is important.

```
<linus> But recency ordering is totally useless for deciding how
    to actually generate the deltas, so the delta ordering is
    something else.


The delta ordering is (wait for it):
- first sort by the "basename" of the object, as defined by
  the name the object was _first_ reached through when
  generating the object list
- within the same basename, sort by size of the object
- but always sort different types separately (commits first).


That's not exactly it, but it's very close.


<njs`> The "_first_ reached" thing is not too important, just you
    need some way to break ties since the same objects may be
    reachable many ways, yes?
```

And as if to clarify:

```
<linus> The point is that it's all really just any random
    heuristic, and the ordering is totally unimportant for
    correctness, but it helps a lot if the heuristic gives
    "clumping" for things that are likely to delta well against
    each other.
```

It is an important point, so secretly, I did my own research and have included my results below. To be fair, it has changed some over time. And through the magic of Revisionistic History, I draw upon this entry from The Git IRC Logs on my father's birthday, March 1:

```
<gitster> The quote from the above linus should be rewritten a
    bit (wait for it):
    - first sort by type.  Different objects never delta with
      each other.
    - then sort by filename/dirname.  hash of the basename
      occupies the top BITS_PER_INT-DIR_BITS bits, and bottom
      DIR_BITS are for the hash of leading path elements.
    - then if we are doing "thin" pack, the objects we are _not_
      going to pack but we know about are sorted earlier than
      other objects.
    - and finally sort by size, larger to smaller.
```

In one swell-foop, clarification and obscurification! Nonetheless, authoritative. Cryptic, yet concise. It even solicits notions of quotes from The Source Code. Clearly, more study is needed.

```
<gitster> That's the sort order.  What this means is:
    - we do not delta different object types.
    - we prefer to delta the objects with the same full path, but
      allow files with the same name from different directories.
    - we always prefer to delta against objects we are not going
      to send, if there are some.
    - we prefer to delta against larger objects, so that we have
      lots of removals.


The penultimate rule is for "thin" packs.  It is used when
the other side is known to have such objects.
```

There it is again. "Thin" packs. I'm thinking to myself, "What is a *thin* pack?" So I ask:

```
<jdl> What is a "thin" pack?


<gitster> Use of --objects-edge to rev-list as the upstream of
    pack-objects.  The pack transfer protocol negotiates that.
```

Woo hoo! Cleared that *right* up!

```
<gitster> There are two directions - push and fetch.
```

There! Did you see it? It is not *"push" and "pull"*! How often the confusion has started here. So casually mentioned, too!

```
<gitster> For push, git-send-pack invokes git-receive-pack on the
    other end.  The receive-pack says "I have up to these commits".
    send-pack looks at them, and computes what are missing from
    the other end.  So "thin" could be the default there.


In the other direction, fetch, git-fetch-pack and
git-clone-pack invokes git-upload-pack on the other end
(via ssh or by talking to the daemon).


There are two cases: fetch-pack with -k and clone-pack is one,
fetch-pack without -k is the other.  clone-pack and fetch-pack
with -k will keep the downloaded packfile without expanded, so
we do not use thin pack transfer.  Otherwise, the generated
pack will have delta without base object in the same pack.


But fetch-pack without -k will explode the received pack into
individual objects, so we automatically ask upload-pack to
give us a thin pack if upload-pack supports it.
```

OK then.

Uh.

Let's return to the previous conversation still in progress.

```
<njs`> and "basename" means something like "the tail of end of
    path of file objects and dir objects, as per basename(3), and
    we just declare all commit and tag objects to have the same
    basename" or something?
```

Luckily, that too is a point that gitster clarified for us!

If I might add, the trick is to make files that *might* be similar be located close to each other in the hash buckets based on their file names. It used to be that "foo/Makefile", "bar/baz/quux/Makefile" and "Makefile" all landed in the same bucket due to their common basename, "Makefile". However, now they land in "close" buckets.

The algorithm allows not just for the *same* bucket, but for *close* buckets to be considered delta candidates. The rationale is essentially that files, like Makefiles, often have very similar content no matter what directory they live in.

```
<linus> I played around with different delta algorithms, and with
    making the "delta window" bigger, but having too big of a
    sliding window makes it very expensive to generate the pack:
    you need to compare every object with a _ton_ of other objects.


There are a number of other trivial heuristics too, which
basically boil down to "don't bother even trying to delta this
pair" if we can tell before-hand that the delta isn't worth it
(due to size differences, where we can take a previous delta
result into account to decide that "ok, no point in trying
that one, it will be worse").


End result: packing is actually very size efficient. It's
somewhat CPU-wasteful, but on the other hand, since you're
really only supposed to do it maybe once a month (and you can
do it during the night), nobody really seems to care.
```

Nice Engineering Touch, there. Find when it doesn't matter, and proclaim it a non-issue. Good style too!

```
<njs`> So, just to repeat to see if I'm following, we start by
    getting a list of the objects we want to pack, we sort it by
    this heuristic (basically lexicographically on the tuple
    (type, basename, size)).


Then we walk through this list, and calculate a delta of
each object against the last n (tunable parameter) objects,
and pick the smallest of these deltas.
```

Vastly simplified, but the essence is there!

```
<linus> Correct.


<njs`> And then once we have picked a delta or fulltext to
    represent each object, we re-sort by recency, and write them
    out in that order.


<linus> Yup. Some other small details:
```

And of course there is the "Other Shoe" Factor too.

```
<linus> - We limit the delta depth to another magic value (right
    now both the window and delta depth magic values are just "10")
```

```
<njs`> Hrm, my intuition is that you'd end up with really _bad_ IO
    patterns, because the things you want are near by, but to
    actually reconstruct them you may have to jump all over in
    random ways.
```

```
<linus> - When we write out a delta, and we haven't yet written
    out the object it is a delta against, we write out the base
    object first.  And no, when we reconstruct them, we actually
    get nice IO patterns, because:
    - larger objects tend to be "more recent" (Linus' law: files grow)
    - we actively try to generate deltas from a larger object to a
      smaller one
    - this means that the top-of-tree very seldom has deltas
      (i.e. deltas in _practice_ are "backwards deltas")
```

Again, we should reread that whole paragraph. Not just because Linus has slipped Linus's Law in there on us, but because it is important. Let's make sure we clarify some of the points here:

```
<njs`> So the point is just that in practice, delta order and
    recency order match each other quite well.
```

```
<linus> Yes. There's another nice side to this (and yes, it was
    designed that way ;):
    - the reason we generate deltas against the larger object is
      actually a big space saver too!
```

```
<njs`> Hmm, but your last comment (if "we haven't yet written out
    the object it is a delta against, we write out the base object
    first"), seems like it would make these facts mostly
    irrelevant because even if in practice you would not have to
    wander around much, in fact you just brute-force say that in
    the cases where you might have to wander, don't do that :-)
```

```
<linus> Yes and no. Notice the rule: we only write out the base
    object first if the delta against it was more recent.  That
    means that you can actually have deltas that refer to a base
    object that is _not_ close to the delta object, but that only
    happens when the delta is needed to generate an _old_ object.
```

```
<linus> See?
```

Yeah, no. I missed that on the first two or three readings myself.

```
<linus> This keeps the front of the pack dense. The front of the
    pack never contains data that isn't relevant to a "recent"
    object.  The size optimization comes from our use of xdelta
    (but is true for many other delta algorithms): removing data
    is cheaper (in size) than adding data.
```

```
When you remove data, you only need to say "copy bytes n--m".
In contrast, in a delta that _adds_ data, you have to say "add
these bytes: 'actual data goes here'"
```

- njs` has quit: Read error: 104 (Connection reset by peer)

  ```
  <linus> Uhhuh. I hope I didn't blow njs` mind.
  ```

- njs` has joined channel #git

  ```
  <pasky> :)
  ```

The silent observers are amused. Of course.

And as if njs` was expected to be omniscient:

```
<linus> njs - did you miss anything?
```

OK, I'll spell it out. That's Geek Humor. If njs` was not actually connected for a little bit there, how would he know if missed anything while he was disconnected? He's a benevolent dictator with a sense of humor! Well noted!

```
<njs`> Stupid router.  Or gremlins, or whatever.
```

It's a cheap shot at Cisco. Take 'em when you can.

```
<njs`> Yes and no. Notice the rule: we only write out the base
    object first if the delta against it was more recent.
```

```
I'm getting lost in all these orders, let me re-read :-)
So the write-out order is from most recent to least recent?
(Conceivably it could be the opposite way too, I'm not sure if
we've said) though my connection back at home is logging, so I
can just read what you said there :-)
```

And for those of you paying attention, the Omniscient Trick has just been detailed!

```
<linus> Yes, we always write out most recent first


<njs`> And, yeah, I got the part about deeper-in-history stuff
    having worse IO characteristics, one sort of doesn't care.


<linus> With the caveat that if the "most recent" needs an older
    object to delta against (hey, shrinking sometimes does
    happen), we write out the old object with the delta.


<njs`> (if only it happened more...)


<linus> Anyway, the pack-file could easily be denser still, but
    because it's used both for streaming (the Git protocol) and
    for on-disk, it has a few pessimizations.
```

Actually, it is a made-up word. But it is a made-up word being used as setup for a later optimization, which is a real word:

```
<linus> In particular, while the pack-file is then compressed,
    it's compressed just one object at a time, so the actual
    compression factor is less than it could be in theory. But it
    means that it's all nice random-access with a simple index to
    do "object name->location in packfile" translation.


<njs`> I'm assuming the real win for delta-ing large->small is
    more homogeneous statistics for gzip to run over?


(You have to put the bytes in one place or another, but
putting them in a larger blob wins on compression)


Actually, what is the compression strategy -- each delta
individually gzipped, the whole file gzipped, somewhere in
between, no compression at all, ....?


Right.
```

Reality IRC sets in. For example:

```
<pasky> I'll read the rest in the morning, I really have to go
    sleep or there's no hope whatsoever for me at the today's
    exam... g'nite all.
```

Heh.

```
<linus> pasky: g'nite


<njs`> pasky: 'luck


<linus> Right: large->small matters exactly because of compression
    behaviour. If it was non-compressed, it probably wouldn't make
    any difference.


<njs`> yeah


<linus> Anyway: I'm not even trying to claim that the pack-files
    are perfect, but they do tend to have a nice balance of
    density vs ease-of use.
```

Gasp! OK, saved. That's a fair Engineering trade off. Close call! In fact, Linus reflects on some Basic Engineering Fundamentals, design options, etc.

```
<linus> More importantly, they allow Git to still _conceptually_
    never deal with deltas at all, and be a "whole object" store.


Which has some problems (we discussed bad huge-file
behaviour on the Git lists the other day), but it does mean
that the basic Git concepts are really really simple and
straightforward.


It's all been quite stable.


Which I think is very much a result of having very simple
basic ideas, so that there's never any confusion about what's
going on.


Bugs happen, but they are "simple" bugs. And bugs that
actually get some object store detail wrong are almost always
```

```
so obvious that they never go anywhere.

<njs`> Yeah.
```

Nuff said.

```
<linus> Anyway.  I'm off for bed. It's not 6AM here, but I've got
     three kids, and have to get up early in the morning to send
     them off. I need my beauty sleep.

<njs`> :-)

<njs`> appreciate the infodump, I really was failing to find the
    details on Git packs :-)
```

And now you know the rest of the story.

Last updated 2014-11-27 19:56:10 CET

Packfile transfer protocols

Git supports transferring data in packfiles over the ssh://, git:// 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. All three transports (ssh, git, file) use the same protocol to transfer data.

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.

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.

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.

```
0032git-upload-pack /project.git\0host=myserver.com\0
```

```
git-proto-request = request-command SP pathname NUL [ host-parameter NUL ]
request-command   = "git-upload-pack" / "git-receive-pack" /
                    "git-upload-archive"   ; case sensitive
pathname          = *( %x01-ff ) ; exclude NUL
host-parameter    = "host=" hostname [ ":" port ]
```

Only host-parameter is allowed in the git-proto-request. Clients MUST NOT attempt to send additional parameters. It 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 \
  "0039git-upload-pack /schacon/gitbook.git\0host=example.com\0" |
  nc -v example.com 9418
```

If the server refuses the request for some reasons, it could abort gracefully with an error message.

```
  error-line     =  PKT-LINE("ERR" SP explanation-text)
```

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, its 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'"
```

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 listing of each reference it has (all branches and tags) along with the object name that each reference currently points to.

```
$ echo -e -n "0039git-upload-pack /schacon/gitbook.git\0host=example.com\0" |
   nc -v example.com 9418
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
```

Server SHOULD terminate each non-flush line using LF ("\n") terminator; client MUST NOT complain if there is no terminator.

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  =  (no-refs / list-of-refs)
                    *shallow
                    flush-pkt

no-refs          =  PKT-LINE(zero-id SP "capabilities^{}"
                    NUL capability-list LF)

list-of-refs     =  first-ref *other-ref
first-ref        =  PKT-LINE(obj-id SP refname
                    NUL capability-list LF)

other-ref        =  PKT-LINE(other-tip / other-peeled)
other-tip        =  obj-id SP refname LF
other-peeled     =  obj-id SP refname "^{}" LF

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
                     flush-pkt

want-list         =  first-want
                     *additional-want

shallow-line      =  PKT-LINE("shallow" SP obj-id)

depth-request     =  PKT-LINE("deepen" SP depth)

first-want        =  PKT-LINE("want" SP obj-id SP capability-list LF)
additional-want   =  PKT-LINE("want" SP obj-id LF)

depth             =  1*DIGIT
```

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.

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 start 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 LF)
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 *NACK* 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.

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 LF)
ack_status      = "continue" / "common" / "ready"
ack             = PKT-LINE("ACK SP obj-id LF)
nak             = PKT-LINE("NAK" LF)
```

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 pack-format.txt 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*, *delete-refs* and *ofs-delta*.

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 and then the packfile that should contain all the objects that the server will need to complete the new references.

```
update-request    =  *shallow ( command-list | push-cert ) [pack-file]

shallow           =  PKT-LINE("shallow" SP obj-id LF)

command-list      =  PKT-LINE(command NUL capability-list LF)
                     *PKT-LINE(command LF)
                     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(LF)
                    *PKT-LINE(command LF)
                    *PKT-LINE(gpg-signature-lines LF)
                    PKT-LINE("push-cert-end" LF)

pack-file         = "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 pack-file MUST NOT be sent if the only command used is *delete*.

A pack-file 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 pack-file. 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.

Currently, the following header fields are defined:

`pusher` ident
: Identify the GPG key in "Human Readable Name <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* 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 LF)
unpack-result     = "ok" / error-msg

command-status    = command-ok / command-fail
```

```
  command-ok         = PKT-LINE("ok" SP refname LF)
  command-fail       = PKT-LINE("ng" SP refname SP error-msg LF)

  error-msg          = 1*(OCTECT) ; where not "ok"
```

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: 007c74730d410fcb6603ace96f1dc55ea6196122532d refs/heads/local\0report-status delete-refs ofs-delta\n
   S: 003e7d1665144a3a975c05f1f43902ddaf084e784dbe refs/heads/debug\n
   S: 003f74730d410fcb6603ace96f1dc55ea6196122532d refs/heads/master\n
   S: 003f74730d410fcb6603ace96f1dc55ea6196122532d refs/heads/team\n
   S: 0000

   C: 003e7d1665144a3a975c05f1f43902ddaf084e784dbe 74730d410fcb6603ace96f1dc55ea6196122532d refs/heads/debug
   C: 003e74730d410fcb6603ace96f1dc55ea6196122532d 5a3f6be755bbb7deae50065988cbfa1ffa9ab68a refs/heads/maste
   C: 0000
   C: [PACKDATA]

   S: 000eunpack ok\n
   S: 0018ok refs/heads/debug\n
   S: 002ang refs/heads/master non-fast-forward\n
```

Last updated 2014-11-27 19:58:08 CET

Git Protocol Capabilities

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 was 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*, *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* capability 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 that the server hasn't found a common base on 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 client to better understand the server's in-memory state. See pack-protocol.txt, 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 server can send, and client 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 maximum of one of "side-band" and "side- band-64k". Server MUST diagnose it as an error if client requests both.

ofs-delta

Server can send, and client 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.

shallow

This capability adds "deepen", "shallow" and "unshallow" commands to the fetch-pack/upload-pack protocol so clients can request shallow clones.

no-progress

The client was started with "git clone -q" or something, 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 referrant 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 pack-protocol.txt for example messages.

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.

allow-tip-sha1-in-want

If the upload-pack server advertises this capability, fetch-pack may send "want" lines with SHA-1s that exist at the server but are not advertised by upload-pack.

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.

Last updated 2015-03-26 21:44:44 CET

Documentation Common to Pack and Http Protocols

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.

The maximum length of a pkt-line's data component is 65520 bytes. Implementations MUST NOT send pkt-line whose length exceeds 65524 (65520 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"            ""
```

Last updated 2014-11-27 19:57:04 CET

Use of index and Racy Git problem

Background

The index is one of the most important data structures in Git. It represents a virtual working tree state by recording list of paths and their object names and serves as a staging area to write out the next tree object to be committed. The state is "virtual" in the sense that it does not necessarily have to, and often does not, match the files in the working tree.

There are cases Git needs to examine the differences between the virtual working tree state in the index and the files in the working tree. The most obvious case is when the user asks `git diff` (or its low level implementation, `git diff-files`) or `git-ls-files --modified`. In addition, Git internally checks if the files in the working tree are different from what are recorded in the index to avoid stomping on local changes in them during patch application, switching branches, and merging.

In order to speed up this comparison between the files in the working tree and the index entries, the index entries record the information obtained from the filesystem via `lstat(2)` system call when they were last updated. When checking if they differ, Git first runs `lstat(2)` on the files and compares the result with this information (this is what was originally done by the `ce_match_stat()` function, but the current code does it in `ce_match_stat_basic()` function). If some of these "cached stat information" fields do not match, Git can tell that the files are modified without even looking at their contents.

Note: not all members in `struct stat` obtained via `lstat(2)` are used for this comparison. For example, `st_atime` obviously is not useful. Currently, Git compares the file type (regular files vs symbolic links) and executable bits (only for regular files) from `st_mode` member, `st_mtime` and `st_ctime` timestamps, `st_uid`, `st_gid`, `st_ino`, and `st_size` members. With a `USE_STDEV` compile-time option, `st_dev` is also compared, but this is not enabled by default because this member is not stable on network filesystems. With `USE_NSEC` compile-time option, `st_mtim.tv_nsec` and `st_ctim.tv_nsec` members are also compared, but this is not enabled by default because in-core timestamps can have finer granularity than on-disk timestamps, resulting in meaningless changes when an inode is evicted from the inode cache. See commit 8ce13b0 of git://git.kernel.org/pub/scm/linux/kernel/git/tglx/history.git ([PATCH] Sync in core time granularity with filesystems, 2005-01-04).

Racy Git

There is one slight problem with the optimization based on the cached stat information. Consider this sequence:

```
: modify 'foo'
$ git update-index 'foo'
: modify 'foo' again, in-place, without changing its size
```

The first `update-index` computes the object name of the contents of file `foo` and updates the index entry for `foo` along with the `struct stat` information. If the modification that follows it happens very fast so that the file's `st_mtime` timestamp does not change, after this sequence, the cached stat information the index entry records still exactly match what you would see in the filesystem, even though the file `foo` is now different. This way, Git can incorrectly think files in the working tree are unmodified even though they actually are. This is called the "racy Git" problem (discovered by Pasky), and the entries that appear clean when they may not be because of this problem are called "racily clean".

To avoid this problem, Git does two things:

1. When the cached stat information says the file has not been modified, and the `st_mtime` is the same as (or newer than) the timestamp of the index file itself (which is the time `git update-index foo` finished running in the above example), it also compares the contents with the object registered in the index entry to make sure they match.
2. When the index file is updated that contains racily clean entries, cached `st_size` information is truncated to zero before writing a new version of the index file.

Because the index file itself is written after collecting all the stat information from updated paths, `st_mtime` timestamp of it is usually the same as or newer than any of the paths the index contains. And no matter how quick the modification that follows `git update-index foo` finishes, the resulting `st_mtime` timestamp on `foo` cannot get a value earlier than the index file. Therefore, index entries that can be racily clean are limited to the ones that have the same timestamp as the index file itself.

The callers that want to check if an index entry matches the corresponding file in the working tree continue to call `ce_match_stat()`, but with this change, `ce_match_stat()` uses `ce_modified_check_fs()` to see if racily clean ones are actually clean after comparing the cached stat information using `ce_match_stat_basic()`.

The problem the latter solves is this sequence:

```
$ git update-index 'foo'
: modify 'foo' in-place without changing its size
: wait for enough time
$ git update-index 'bar'
```

Without the latter, the timestamp of the index file gets a newer value, and falsely clean entry `foo` would not be caught by the timestamp comparison check done with the former logic anymore. The latter makes sure that the cached stat information for `foo` would never match with the file in the working tree, so later checks by `ce_match_stat_basic()` would report that the index entry does not match the file and Git does not have to fall back on more expensive `ce_modified_check_fs()`.

Runtime penalty

The runtime penalty of falling back to `ce_modified_check_fs()` from `ce_match_stat()` can be very expensive when there are many racily clean entries. An obvious way to artificially create this situation is to give the same timestamp to all the files in the working tree in a large project, run `git update-index` on them, and give the same timestamp to the index file:

```
$ date >.datestamp
$ git ls-files | xargs touch -r .datestamp
$ git ls-files | git update-index --stdin
$ touch -r .datestamp .git/index
```

This will make all index entries racily clean. The linux project, for example, there are over 20,000 files in the working tree. On my Athlon 64 X2 3800+, after the above:

```
$ /usr/bin/time git diff-files
1.68user 0.54system 0:02.22elapsed 100%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+67111minor)pagefaults 0swaps
$ git update-index MAINTAINERS
$ /usr/bin/time git diff-files
0.02user 0.12system 0:00.14elapsed 100%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+935minor)pagefaults 0swaps
```

Running `git update-index` in the middle checked the racily clean entries, and left the cached `st_mtime` for all the paths intact because they were actually clean (so this step took about the same amount of time as the first `git diff-files`). After that, they are not racily clean anymore but are truly clean, so the second invocation of `git diff-files` fully took advantage of the cached stat information.

Avoiding runtime penalty

In order to avoid the above runtime penalty, post 1.4.2 Git used to have a code that made sure the index file got timestamp newer than the youngest files in the index when there are many young files with the same timestamp as the resulting index file would otherwise would have by waiting before finishing writing the index file out.

I suspected that in practice the situation where many paths in the index are all racily clean was quite rare. The only code paths that can record recent timestamp for large number of paths are:

1. Initial `git add .` of a large project.
2. `git checkout` of a large project from an empty index into an unpopulated working tree.

Note: switching branches with `git checkout` keeps the cached stat information of existing working tree files that are the same between the current branch and the new branch, which are all older than the resulting index file, and they will not become racily clean. Only the files that are actually checked out can become racily clean.

In a large project where raciness avoidance cost really matters, however, the initial computation of all object names in the index takes more than one second, and the index file is written out after all that happens. Therefore the timestamp of the index file will be more than one seconds later than the youngest file in the working tree. This means that in these cases there actually will not be any racily clean entry in the resulting index.

Based on this discussion, the current code does not use the "workaround" to avoid the runtime penalty that does not exist in practice anymore. This was done with commit 0fc82cff on Aug 15, 2006.

Last updated 2014-11-27 19:56:10 CET

Git-send-pack internals

Overall operation

1. Connects to the remote side and invokes git-receive-pack.
2. Learns what refs the remote has and what commit they point at. Matches them to the refspecs we are pushing.
3. Checks if there are non-fast-forwards. Unlike fetch-pack, the repository send-pack runs in is supposed to be a superset of the recipient in fast-forward cases, so there is no need for want/have exchanges, and fast-forward check can be done locally. Tell the result to the other end.
4. Calls pack_objects() which generates a packfile and sends it over to the other end.
5. If the remote side is new enough (v1.1.0 or later), wait for the unpack and hook status from the other end.
6. Exit with appropriate error codes.

Pack_objects pipeline

This function gets one file descriptor (`fd`) which is either a socket (over the network) or a pipe (local). What's written to this fd goes to git-receive-pack to be unpacked.

```
send-pack ---> fd ---> receive-pack
```

The function pack_objects creates a pipe and then forks. The forked child execs pack-objects with --revs to receive revision parameters from its standard input. This process will write the packfile to the other end.

```
send-pack
   |
   pack_objects() ---> fd ---> receive-pack
      | ^ (pipe)
      v |
     (child)
```

The child dup2's to arrange its standard output to go back to the other end, and read its standard input to come from the pipe. After that it exec's pack-objects. On the other hand, the parent process, before starting to feed the child pipeline, closes the reading side of the pipe and fd to receive-pack.

```
send-pack
   |
   pack_objects(parent)
      |
      v [0]
     pack-objects [0] ---> receive-pack
```

[jc: the pipeline was much more complex and needed documentation before I understood an earlier bug, but now it is trivial and straightforward.]

Last updated 2014-01-25 09:03:55 CET

Shallow commits

Definition

Shallow commits do have parents, but not in the shallow repo, and therefore grafts are introduced pretending that these commits have no parents.

The basic idea is to write the SHA-1s of shallow commits into $GIT_DIR/shallow, and handle its contents like the contents of $GIT_DIR/info/grafts (with the difference that shallow cannot contain parent information).

This information is stored in a new file instead of grafts, or even the config, since the user should not touch that file at all (even throughout development of the shallow clone, it was never manually edited!).

Each line contains exactly one SHA-1. When read, a commit_graft will be constructed, which has nr_parent < 0 to make it easier to discern from user provided grafts.

Since fsck-objects relies on the library to read the objects, it honours shallow commits automatically.

There are some unfinished ends of the whole shallow business:

- maybe we have to force non-thin packs when fetching into a shallow repo (ATM they are forced non-thin).
- A special handling of a shallow upstream is needed. At some stage, upload-pack has to check if it sends a shallow commit, and it should send that information early (or fail, if the client does not support shallow repositories). There is no support at all for this in this patch series.
- Instead of locking $GIT_DIR/shallow at the start, just the timestamp of it is noted, and when it comes to writing it, a check is performed if the mtime is still the same, dying if it is not.
- It is unclear how "push into/from a shallow repo" should behave.
- If you deepen a history, you’d want to get the tags of the newly stored (but older!) commits. This does not work right now.

To make a shallow clone, you can call "git-clone --depth 20 repo". The result contains only commit chains with a length of at most 20. It also writes an appropriate $GIT_DIR/shallow.

You can deepen a shallow repository with "git-fetch --depth 20 repo branch", which will fetch branch from repo, but stop at depth 20, updating $GIT_DIR/shallow.

The special depth 2147483647 (or 0x7fffffff, the largest positive number a signed 32-bit integer can contain) means infinite depth.

Last updated 2014-11-27 19:55:05 CET

Submitting Patches

Here are some guidelines for people who want to contribute their code to this software.

(0) Decide what to base your work on.

In general, always base your work on the oldest branch that your change is relevant to.

- A bugfix should be based on *maint* in general. If the bug is not present in *maint*, base it on *master*. For a bug that’s not yet in *master*, find the topic that introduces the regression, and base your work on the tip of the topic.
- A new feature should be based on *master* in general. If the new feature depends on a topic that is in *pu*, but not in *master*, base your work on the tip of that topic.
- Corrections and enhancements to a topic not yet in *master* should be based on the tip of that topic. If the topic has not been merged to *next*, it’s alright to add a note to squash minor corrections into the series.
- In the exceptional case that a new feature depends on several topics not in *master*, start working on *next* or *pu* privately and send out patches for discussion. Before the final merge, you may have to wait until some of the dependent topics graduate to *master*, and rebase your work.
- Some parts of the system have dedicated maintainers with their own repositories (see the section "Subsystems" below). Changes to these parts should be based on their trees.

To find the tip of a topic branch, run "git log --first-parent master..pu" and look for the merge commit. The second parent of this commit is the tip of the topic branch.

(1) Make separate commits for logically separate changes.

Unless your patch is really trivial, you should not be sending out a patch that was generated between your working tree and your commit head. Instead, always make a commit with complete commit message and generate a series of patches from your repository. It is a good discipline.

Give an explanation for the change(s) that is detailed enough so that people can judge if it is good thing to do, without reading the actual patch text to determine how well the code does what the explanation promises to do.

If your description starts to get too long, that's a sign that you probably need to split up your commit to finer grained pieces. That being said, patches which plainly describe the things that help reviewers check the patch, and future maintainers understand the code, are the most beautiful patches. Descriptions that summarise the point in the subject well, and describe the motivation for the change, the approach taken by the change, and if relevant how this differs substantially from the prior version, are all good things to have.

Make sure that you have tests for the bug you are fixing. See t/README for guidance.

When adding a new feature, make sure that you have new tests to show the feature triggers the new behaviour when it should, and to show the feature does not trigger when it shouldn't. Also make sure that the test suite passes after your commit. Do not forget to update the documentation to describe the updated behaviour.

Speaking of the documentation, it is currently a liberal mixture of US and UK English norms for spelling and grammar, which is somewhat unfortunate. A huge patch that touches the files all over the place only to correct the inconsistency is not welcome, though. Potential clashes with other changes that can result from such a patch are not worth it. We prefer to gradually reconcile the inconsistencies in favor of US English, with small and easily digestible patches, as a side effect of doing some other real work in the vicinity (e.g. rewriting a paragraph for clarity, while turning en_UK spelling to en_US). Obvious typographical fixes are much more welcomed ("teh → "the"), preferably submitted as independent patches separate from other documentation changes.

Oh, another thing. We are picky about whitespaces. Make sure your changes do not trigger errors with the sample pre-commit hook shipped in templates/hooks—pre-commit. To help ensure this does not happen, run git diff --check on your changes before you commit.

(2) Describe your changes well.

The first line of the commit message should be a short description (50 characters is the soft limit, see DISCUSSION in git-commit(1)), and should skip the full stop. It is also conventional in most cases to prefix the first line with "area: " where the area is a filename or identifier for the general area of the code being modified, e.g.

1. archive: ustar header checksum is computed unsigned
2. git-cherry-pick.txt: clarify the use of revision range notation

If in doubt which identifier to use, run "git log --no-merges" on the files you are modifying to see the current conventions.

The body should provide a meaningful commit message, which:

1. explains the problem the change tries to solve, iow, what is wrong with the current code without the change.
2. justifies the way the change solves the problem, iow, why the result with the change is better.
3. alternate solutions considered but discarded, if any.

Describe your changes in imperative mood, e.g. "make xyzzy do frotz" instead of "[This patch] makes xyzzy do frotz" or "[I] changed xyzzy to do frotz", as if you are giving orders to the codebase to change its behaviour. Try to make sure your explanation can be understood without external resources. Instead of giving a URL to a mailing list archive, summarize the relevant points of the discussion.

(3) Generate your patch using Git tools out of your commits.

Git based diff tools generate unidiff which is the preferred format.

You do not have to be afraid to use -M option to "git diff" or "git format-patch", if your patch involves file renames. The receiving end can handle them just fine.

Please make sure your patch does not add commented out debugging code, or include any extra files which do not relate to what your patch is trying to achieve. Make sure to review your patch after generating it, to ensure accuracy. Before sending out, please make sure it cleanly applies to the "master" branch head. If you are preparing a work based on "next" branch, that is fine, but please mark it as such.

(4) Sending your patches.

Learn to use format-patch and send-email if possible. These commands are optimized for the workflow of sending patches, avoiding many ways your existing e-mail client that is optimized for "multipart/*" mime type e-mails to corrupt and render your patches unusable.

People on the Git mailing list need to be able to read and comment on the changes you are submitting. It is important for a developer to be able to "quote" your changes, using standard e-mail tools, so that they may comment on specific portions of your code. For this reason, each patch should be submitted "inline" in a separate message.

Multiple related patches should be grouped into their own e-mail thread to help readers find all parts of the series. To that end, send them as replies to either an additional "cover letter" message (see below), the first patch, or the respective preceding patch.

If your log message (including your name on the Signed-off-by line) is not writable in ASCII, make sure that you send off a message in the correct encoding.

Warning Be wary of your MUAs word-wrap corrupting your patch. Do not cut-n-paste your patch; you can lose tabs that way if you are not careful.

It is a common convention to prefix your subject line with [PATCH]. This lets people easily distinguish patches from other e-mail discussions. Use of additional markers after PATCH and the closing bracket to mark the nature of the patch is also encouraged. E.g. [PATCH/RFC] is often used when the patch is not ready to be applied but it is for discussion, [PATCH v2], [PATCH v3] etc. are often seen when you are sending an update to what you have previously sent.

"git format-patch" command follows the best current practice to format the body of an e-mail message. At the beginning of the patch should come your commit message, ending with the Signed-off-by: lines, and a line that consists of three dashes, followed by the diffstat information and the patch itself. If you are forwarding a patch from somebody else, optionally, at the beginning of the e-mail message just before the commit message starts, you can put a "From: " line to name that person.

You often want to add additional explanation about the patch, other than the commit message itself. Place such "cover letter" material between the three-dash line and the diffstat. For patches requiring multiple iterations of review and discussion, an explanation of changes between each iteration can be kept in Git-notes and inserted automatically following the three-dash line via `git format-patch --notes`.

Do not attach the patch as a MIME attachment, compressed or not. Do not let your e-mail client send quoted-printable. Do not let your e-mail client send format=flowed which would destroy whitespaces in your patches. Many popular e-mail applications will not always transmit a MIME attachment as plain text, making it impossible to comment on your code. A MIME attachment also takes a bit more time to process. This does not decrease the likelihood of your MIME-attached change being accepted, but it makes it more likely that it will be postponed.

Exception: If your mailer is mangling patches then someone may ask you to re-send them using MIME, that is OK.

Do not PGP sign your patch, at least for now. Most likely, your maintainer or other people on the list would not have your PGP key and would not bother obtaining it anyway. Your patch is not judged by who you are; a good patch from an unknown origin has a far better chance of being accepted than a patch from a known, respected origin that is done poorly or does incorrect things.

If you really really really really want to do a PGP signed patch, format it as "multipart/signed", not a text/plain message that starts with *-----BEGIN PGP SIGNED MESSAGE-----*. That is not a text/plain, it’s something else.

Send your patch with "To:" set to the mailing list, with "cc:" listing people who are involved in the area you are touching (the output from "git blame $path" and "git shortlog --no-merges $path" would help to identify them), to solicit comments and reviews.

After the list reached a consensus that it is a good idea to apply the patch, re-send it with "To:" set to the maintainer [**1**] and "cc:" the list [**2**] for inclusion.

Do not forget to add trailers such as "Acked-by:", "Reviewed-by:" and "Tested-by:" lines as necessary to credit people who helped your patch.

```
[Addresses]
 *1* The current maintainer: gitster@pobox.com
 *2* The mailing list: git@vger.kernel.org
```

(5) Sign your work

To improve tracking of who did what, we’ve borrowed the "sign-off" procedure from the Linux kernel project on patches that are being emailed around. Although core Git is a lot smaller project it is a good discipline to follow it.

The sign-off is a simple line at the end of the explanation for the patch, which certifies that you wrote it or otherwise have the right to pass it on as a open-source patch. The rules are pretty simple: if you can certify the below:

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.
```

```
(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

then you just add a line saying

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

This line can be automatically added by Git if you run the git-commit command with the -s option.

Notice that you can place your own Signed-off-by: line when forwarding somebody else’s patch with the above rules for D-C-O. Indeed you are encouraged to do so. Do not forget to place an in-body "From: " line at the beginning to properly attribute the change to its true author (see (2) above).

Also notice that a real name is used in the Signed-off-by: line. Please don’t hide your real name.

If you like, you can put extra tags at the end:

1. "Reported-by:" is used to credit someone who found the bug that the patch attempts to fix.
2. "Acked-by:" says that the person who is more familiar with the area the patch attempts to modify liked the patch.
3. "Reviewed-by:", unlike the other tags, can only be offered by the reviewer and means that she is completely satisfied that the patch is ready for application. It is usually offered only after a detailed review.
4. "Tested-by:" is used to indicate that the person applied the patch and found it to have the desired effect.

You can also create your own tag or use one that’s in common usage such as "Thanks-to:", "Based-on-patch-by:", or "Mentored-by:".

```
Subsystems with dedicated maintainers

Some parts of the system have dedicated maintainers with their own
repositories.

 - git-gui/ comes from git-gui project, maintained by Pat Thoyts:

        git://repo.or.cz/git-gui.git

 - gitk-git/ comes from Paul Mackerras's gitk project:

        git://ozlabs.org/~paulus/gitk

 - po/ comes from the localization coordinator, Jiang Xin:

        https://github.com/git-l10n/git-po/

Patches to these parts should be based on their trees.
```

An ideal patch flow

Here is an ideal patch flow for this project the current maintainer suggests to the contributors:

```
(0) You come up with an itch.  You code it up.


(1) Send it to the list and cc people who may need to know about
    the change.


The people who may need to know are the ones whose code you
are butchering.  These people happen to be the ones who are
most likely to be knowledgeable enough to help you, but
they have no obligation to help you (i.e. you ask for help,
don't demand).  "git log -p -- $area_you_are_modifying" would
help you find out who they are.


(2) You get comments and suggestions for improvements.  You may
    even get them in a "on top of your change" patch form.


(3) Polish, refine, and re-send to the list and the people who
    spend their time to improve your patch.  Go back to step (2).


(4) The list forms consensus that the last round of your patch is
    good.  Send it to the maintainer and cc the list.


(5) A topic branch is created with the patch and is merged to 'next',
    and cooked further and eventually graduates to 'master'.
```

In any time between the (2)-(3) cycle, the maintainer may pick it up from the list and queue it to *pu*, in order to make it easier for people play with it without having to pick up and apply the patch to their trees themselves.

Know the status of your patch after submission

- You can use Git itself to find out when your patch is merged in master. *git pull --rebase* will automatically skip already-applied patches, and will let you know. This works only if you rebase on top of the branch in which your patch has been merged (i.e. it will not tell you if your patch is merged in pu if you rebase on top of master).
- Read the Git mailing list, the maintainer regularly posts messages entitled "What's cooking in git.git" and "What's in git.git" giving the status of various proposed changes.

MUA specific hints

Some of patches I receive or pick up from the list share common patterns of breakage. Please make sure your MUA is set up properly not to corrupt whitespaces.

See the DISCUSSION section of git-format-patch(1) for hints on checking your patch by mailing it to yourself and applying with git-am(1).

While you are at it, check the resulting commit log message from a trial run of applying the patch. If what is in the resulting commit is not exactly what you would want to see, it is very likely that your maintainer would end up hand editing the log message when he applies your patch. Things like "Hi, this is my first patch.\n", if you really want to put in the patch e-mail, should come after the three-dash line that signals the end of the commit message.

Pine

(Johannes Schindelin)

I don't know how many people still use pine, but for those poor souls it may be good to mention that the quell-flowed-text is needed for recent versions.

And the "no-strip-whitespace-before-send" option, too. AFAIK it was introduced in 4.60.

(Linus Torvalds)

And 4.58 needs at least this.

```
diff-tree 8326dd8350be64ac7fc805f6563a1d61ad10d32c (from e886a61f76edf5410573e92e38ce22974f9c40f1)
Author: Linus Torvalds <torvalds@g5.osdl.org>
Date:   Mon Aug 15 17:23:51 2005 -0700

    Fix pine whitespace-corruption bug

    There's no excuse for unconditionally removing whitespace from
    the pico buffers on close.

diff --git a/pico/pico.c b/pico/pico.c
--- a/pico/pico.c
+++ b/pico/pico.c
@@ -219,7 +219,9 @@ PICO *pm;
            switch(pico_all_done){      /* prepare for/handle final events */
              case COMP_EXIT :          /* already confirmed */
                packheader();
+#if 0
                stripwhitespace();
+#endif
                c |= COMP_EXIT;
                break;
```

(Daniel Barkalow)

```
> A patch to SubmittingPatches, MUA specific help section for
> users of Pine 4.63 would be very much appreciated.
```

Ah, it looks like a recent version changed the default behavior to do the right thing, and inverted the sense of the configuration option. (Either that or Gentoo did it.) So you need to set the "no-strip-whitespace-before-send" option, unless the option you have is "strip-whitespace-before-send", in which case you should avoid checking it.

Thunderbird, KMail, GMail

See the MUA-SPECIFIC HINTS section of git-format-patch(1).

Gnus

| in the **Summary** buffer can be used to pipe the current message to an external program, and this is a handy way to drive "git am". However, if the message is MIME encoded, what is piped into the program is the representation you see in your **Article** buffer after unwrapping MIME. This is often not what you would want for two reasons. It tends to screw up non ASCII characters (most notably in people's names), and also whitespaces (fatal in patches). Running *C-u g* to display the message in raw form before using | to run the pipe can work this problem around.

Trivial merge rules

This document describes the outcomes of the trivial merge logic in read-tree.

One-way merge

This replaces the index with a different tree, keeping the stat info for entries that don’t change, and allowing -u to make the minimum required changes to the working tree to have it match.

Entries marked + have stat information. Spaces marked * don’t affect the result.

```
index   tree    result
-----------------------
*       (empty) (empty)
(empty) tree    tree
index+  tree    tree
index+  index   index+
```

Two-way merge

It is permitted for the index to lack an entry; this does not prevent any case from applying.

If the index exists, it is an error for it not to match either the old or the result.

If multiple cases apply, the one used is listed first.

A result which changes the index is an error if the index is not empty and not up-to-date.

Entries marked + have stat information. Spaces marked * don’t affect the result.

```
case  index   old     new     result
-------------------------------------
0/2   (empty) *       (empty) (empty)
1/3   (empty) *       new     new
4/5   index+  (empty) (empty) index+
6/7   index+  (empty) index   index+
10    index+  index   (empty) (empty)
14/15 index+  old     old     index+
18/19 index+  old     index   index+
20    index+  index   new     new
```

Three-way merge

It is permitted for the index to lack an entry; this does not prevent any case from applying.

If the index exists, it is an error for it not to match either the head or (if the merge is trivial) the result.

If multiple cases apply, the one used is listed first.

A result of "no merge" means that index is left in stage 0, ancest in stage 1, head in stage 2, and remote in stage 3 (if any of these are empty, no entry is left for that stage). Otherwise, the given entry is left in stage 0, and there are no other entries.

A result of "no merge" is an error if the index is not empty and not up-to-date.

empty means that the tree must not have a directory-file conflict with the entry.

For multiple ancestors, a + means that this case applies even if only one ancestor or remote fits; a ^ means all of the ancestors must be the same.

```
case  ancest    head    remote    result
----------------------------------------
1     (empty)+  (empty) (empty)   (empty)
2ALT  (empty)+  *empty* remote    remote
2     (empty)^  (empty) remote    no merge
3ALT  (empty)+  head    *empty*   head
3     (empty)^  head    (empty)   no merge
4     (empty)^  head    remote    no merge
5ALT  *         head    head      head
6     ancest+   (empty) (empty)   no merge
8     ancest^   (empty) ancest    no merge
7     ancest+   (empty) remote    no merge
10    ancest^   ancest  (empty)   no merge
9     ancest+   head    (empty)   no merge
16    anc1/anc2 anc1    anc2      no merge
13    ancest+   head    ancest    head
14    ancest+   ancest  remote    remote
```

```
11    ancest+   head    remote    no merge
```

Only #2ALT and #3ALT use **empty**, because these are the only cases where there can be conflicts that didn't exist before. Note that we allow directory-file conflicts between things in different stages after the trivial merge.

A possible alternative for #6 is (empty), which would make it like #1. This is not used, due to the likelihood that it arises due to moving the file to multiple different locations or moving and deleting it in different branches.

Case #1 is included for completeness, and also in case we decide to put on + markings; any path that is never mentioned at all isn't handled.

Note that #16 is when both #13 and #14 apply; in this case, we refuse the trivial merge, because we can't tell from this data which is right. This is a case of a reverted patch (in some direction, maybe multiple times), and the right answer depends on looking at crossings of history or common ancestors of the ancestors.

Note that, between #6, #7, #9, and #11, all cases not otherwise covered are handled in this table.

For #8 and #10, there is alternative behavior, not currently implemented, where the result is (empty). As currently implemented, the automatic merge will generally give this effect.

Last updated 2014-01-25 09:03:55 CET

Git installation

Normally you can just do "make" followed by "make install", and that will install the git programs in your own ~/bin/ directory. If you want to do a global install, you can do

```
$ make prefix=/usr all doc info ;# as yourself
# make prefix=/usr install install-doc install-html install-info ;# as root
```

(or prefix=/usr/local, of course). Just like any program suite that uses $prefix, the built results have some paths encoded, which are derived from $prefix, so "make all; make prefix=/usr install" would not work.

The beginning of the Makefile documents many variables that affect the way git is built. You can override them either from the command line, or in a config.mak file.

Alternatively you can use autoconf generated ./configure script to set up install paths (via config.mak.autogen), so you can write instead

```
$ make configure ;# as yourself
$ ./configure --prefix=/usr ;# as yourself
$ make all doc ;# as yourself
# make install install-doc install-html;# as root
```

If you're willing to trade off (much) longer build time for a later faster git you can also do a profile feedback build with

```
$ make prefix=/usr profile
# make prefix=/usr PROFILE=BUILD install
```

This will run the complete test suite as training workload and then rebuild git with the generated profile feedback. This results in a git which is a few percent faster on CPU intensive workloads. This may be a good tradeoff for distribution packagers.

Alternatively you can run profile feedback only with the git benchmark suite. This runs significantly faster than the full test suite, but has less coverage:

```
$ make prefix=/usr profile-fast
# make prefix=/usr PROFILE=BUILD install
```

Or if you just want to install a profile-optimized version of git into your home directory, you could run:

```
$ make profile-install
```

or $ make profile-fast-install

As a caveat: a profile-optimized build takes a **lot** longer since the git tree must be built twice, and in order for the profiling measurements to work properly, ccache must be disabled and the test suite has to be run using only a single CPU. In addition, the profile feedback build stage currently generates a lot of additional compiler warnings.

Issues of note:

- Ancient versions of GNU Interactive Tools (pre-4.9.2) installed a program "git", whose name conflicts with this program. But with version 4.9.2, after long hiatus without active maintenance (since around 1997), it changed its name to gnuit and the name conflict is no longer a problem.
  NOTE: When compiled with backward compatibility option, the GNU Interactive Tools package still can install "git", but you can build it with --disable-transition option to avoid this.
- You can use git after building but without installing if you want to test drive it. Simply run git found in bin-wrappers directory in the build directory, or prepend that directory to your $PATH. This however is less efficient than running an installed git, as you always need an extra fork+exec to run any git subcommand.
  It is still possible to use git without installing by setting a few environment variables, which was the way this was done traditionally. But using git found in bin-wrappers directory in the build directory is far simpler. As a historical reference, the old way went like this:

  ```
  GIT_EXEC_PATH=`pwd`
  PATH=`pwd`:$PATH
  GITPERLLIB=`pwd`/perl/blib/lib
  export GIT_EXEC_PATH PATH GITPERLLIB
  ```

- Git is reasonably self-sufficient, but does depend on a few external programs and libraries. Git can be used without most of them by adding the approriate "NO_<LIBRARY>=YesPlease" to the make command line or config.mak file.
- "zlib", the compression library. Git won't build without it.
- "ssh" is used to push and pull over the net.
- A POSIX-compliant shell is required to run many scripts needed for everyday use (e.g. "bisect", "pull").
- "Perl" version 5.8 or later is needed to use some of the features (e.g. preparing a partial commit using "git add -

i/-p", interacting with svn repositories with "git svn"). If you can live without these, use NO_PERL. Note that recent releases of Redhat/Fedora are reported to ship Perl binary package with some core modules stripped away (see http://lwn.net/Articles/477234/), so you might need to install additional packages other than Perl itself, e.g. Time::HiRes.

- git-imap-send needs the OpenSSL library to talk IMAP over SSL if you are using libcurl older than 7.34.0. Otherwise you can use NO_OPENSSL without losing git-imap-send.
  By default, git uses OpenSSL for SHA1 but it will use its own library (inspired by Mozilla's) with either NO_OPENSSL or BLK_SHA1. Also included is a version optimized for PowerPC (PPC_SHA1).
- "libcurl" library is used by git-http-fetch, git-fetch, and, if the curl version >= 7.34.0, for git-imap-send. You might also want the "curl" executable for debugging purposes. If you do not use http:// or https:// repositories, and do not want to put patches into an IMAP mailbox, you do not have to have them (use NO_CURL).
- "expat" library; git-http-push uses it for remote lock management over DAV. Similar to "curl" above, this is optional (with NO_EXPAT).
- "wish", the Tcl/Tk windowing shell is used in gitk to show the history graphically, and in git-gui. If you don't want gitk or git-gui, you can use NO_TCLTK.
- A gettext library is used by default for localizing Git. The primary target is GNU libintl, but the Solaris gettext implementation also works.
  We need a gettext.h on the system for C code, gettext.sh (or Solaris gettext(1)) for shell scripts, and libintl-perl for Perl programs.
  Set NO_GETTEXT to disable localization support and make Git only use English. Under autoconf the configure script will do this automatically if it can't find libintl on the system.
- Python version 2.4 or later (but not 3.x, which is not supported by Perforce) is needed to use the git-p4 interface to Perforce.
- Some platform specific issues are dealt with Makefile rules, but depending on your specific installation, you may not have all the libraries/tools needed, or you may have necessary libraries at unusual locations. Please look at the top of the Makefile to see what can be adjusted for your needs. You can place local settings in config.mak and the Makefile will include them. Note that config.mak is not distributed; the name is reserved for local settings.
- To build and install documentation suite, you need to have the asciidoc/xmlto toolchain. Because not many people are inclined to install the tools, the default build target ("make all") does *not* build them.
  "make doc" builds documentation in man and html formats; there are also "make man", "make html" and "make info". Note that "make html" requires asciidoc, but not xmlto. "make man" (and thus make doc) requires both.
  "make install-doc" installs documentation in man format only; there are also "make install-man", "make install-html" and "make install-info".
  Building and installing the info file additionally requires makeinfo and docbook2X. Version 0.8.3 is known to work.
  Building and installing the pdf file additionally requires dblatex. Version >= 0.2.7 is known to work.
  All formats require at least asciidoc 8.4.1.
  There are also "make quick-install-doc", "make quick-install-man" and "make quick-install-html" which install preformatted man pages and html documentation. To use these build targets, you need to clone two separate git-htmldocs and git-manpages repositories next to the clone of git itself.
  It has been reported that docbook-xsl version 1.72 and 1.73 are buggy; 1.72 misformats manual pages for callouts, and 1.73 needs the patch in contrib/patches/docbook-xsl-manpages-charmap.patch
  Users attempting to build the documentation on Cygwin may need to ensure that the /etc/xml/catalog file looks something like this:

```
<?xml version="1.0"?>
<!DOCTYPE catalog PUBLIC
   "-//OASIS//DTD Entity Resolution XML Catalog V1.0//EN"
   "http://www.oasis-open.org/committees/entity/release/1.0/catalog.dtd"
>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
  <rewriteURI
    uriStartString = "http://docbook.sourceforge.net/release/xsl/current"
    rewritePrefix = "/usr/share/sgml/docbook/xsl-stylesheets"
  />
  <rewriteURI
    uriStartString="http://www.oasis-open.org/docbook/xml/4.5"
    rewritePrefix="/usr/share/sgml/docbook/xml-dtd-4.5"
  />
</catalog>


This can be achieved with the following two xmlcatalog commands:


xmlcatalog --noout \
  --add rewriteURI \
     http://docbook.sourceforge.net/release/xsl/current \
     /usr/share/sgml/docbook/xsl-stylesheets \
  /etc/xml/catalog


xmlcatalog --noout \
  --add rewriteURI \
      http://www.oasis-open.org/docbook/xml/4.5/xsl/current \
      /usr/share/sgml/docbook/xml-dtd-4.5 \
  /etc/xml/catalog
```

Last updated 2015-05-03 21:16:44 CEST