gittutorial(7) Manual Page

NAME

gittutorial - A tutorial introduction to git (for version 1.5.1 or newer)

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:
#   (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. Tools that turn commits into email, for example, use the first line 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.
- 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), Everyday git, The Git User's Manual

GIT

Part of the git(1) suite.

Last updated 2011-07-23 00:49:30 UTC