Git Pocket Guide

What Is Git?

Git is a tool for tracking changes made to a set of files over time, a task traditionally known as “version control.” Although it is most often used by programmers to coordinate changes to software

source code, and it is especially good at that, you can use Git to track any kind of content at all. Any body of related files evolving over time, which we’ll call a “project,” is a candidate for

using Git. With Git, you can:

* Examine the state of your project at earlier points in time
* Show the differences among various states of the project
* Split the project development into multiple independent

lines, called “branches,” which can evolve separately

* Periodically recombine branches in a process called “merging,”

reconciling the changes made in two or more branches

* Allow many people to work on a project simultaneously,

sharing and combining their work as needed

Git is the technology behind the enormously popular “social coding” website GitHub, which includes many wellknown open source projects.

This is called a “root commit,” and most often, there is only one root commit in a repository—the

initial one created when the repository was started. However, you can introduce multiple root commits if you want; the command git checkout --orphan does this. This incorporates multiple independent

histories into a repository, perhaps in order to collect the contents of previously separate projects (see “Importing Disconnected History” on page 154).

Author versus Committer

Cherry-picking carries forward the author information from the original commit, while adding new committer information. This preserves the identification and origin date of the changes, while indicating that they were applied at another point in the repository at a later date, possibly by a different person.

**$ git log --format=fuller**

Other operations that do this are git rebase and git filterbranch; like git cherry-pick, they too create new commits based on existing ones.

A *tag* serves to distinguish a particular commit by giving it a human-readable name in a namespace reserved for this purpose

There are actually two kinds of tags in Git: “lightweight” and “annotated.” This section refers to annotated tags, which are represented as a separate kind of object in the repository database. A lightweight tag is entirely different; it is simply a name pointing directly to a commit (see the upcoming

section on refs to understand how such names work generally).

Object IDs and SHA-1

Git, on the other hand, assigns object identifiers based on an object’s contents, rather than on its relationship to other objects, using a mathematical technique called a *hash function*.

From this property flow a number of key points:

*Single-instance store*

* Git never stores more than one copy of a file. It can’t—if you add a second copy of the file, it will hash the file contents to find its SHA-1 object ID, look in the database, and find that it’s already there. This is also a consequence of the separation of a file’s contents from its name. Trees map filenames onto blobs in a separate step, to determine the contents of a particular filename at any given commit, but Git does not consider the name or other properties of a file when storing it, only its contents.

*Efficient comparisons*

*Database sharing*

* Git repositories can share their object databases at any level with impunity because there can be no aliasing; the binding between an ID and the content to which it refers is immutable.

Where Objects Live

In a Git repository, objects are stored under *.git/objects*. They may be stored individually as “loose” objects, one per file with pathnames built from their object IDs:

**$ find .git/objects -type f**

They may also be collected into more compact data structures

called “packs,” which appear as paired *.idx* and *.pack* files:

**$ ls .git/objects/pack/**

Refs

Git defines two kinds of references, or named pointers, which it calls “refs”:

* A simple ref, which points directly to an object ID (usually a commit or tag)
* A *symbolic ref* (or *symref*), which points to another ref (either simple or symbolic)

Git uses refs to name things, including commits, branches, and tags. Refs inhabit a hierarchical namespace separated by slashes (as with Unix filenames), starting at refs/. A new repository has at least refs/tags/ and refs/heads/, to hold the names of tags and local branches, respectively. There is also refs/remotes/, holding names referring to other repositories; these contain beneath them the ref namespaces of those repositories, and are used in push and pull operations. For example, when you clone a repository, Git creates a “remote” named origin referring to the

source repository.

There are various defaults, which means that you don’t often have to refer to a ref by its full name; for example, in branch operations, Git implicitly looks in refs/heads/ for the name you give.

These are low-level commands that directly display, change, or delete refs. You don’t ordinarily need these, as Git usually handles refs automatically as part of dealing with the objects they represent, such as branches and tags. If you change refs directly, be sure you know what you’re doing!

git show-ref

Display refs and the objects to which they refer

git symbolic-ref

Deals with symbolic refs specifically

git update-ref

Change the value of a ref

git for-each-ref

Apply an action to a set of refs

Branches

A Git branch is the simplest thing possible: a pointer to a commit, as a ref. Or rather, that is its implementation; the branch itself is defined as all points reachable in the commit graph from the

named commit (the “tip” of the branch). The special ref HEAD determines what branch you are on; if HEAD is a symbolic ref for an existing branch, then you are “on” that branch. If, on the

other hand, HEAD is a simple ref directly naming a commit by its SHA-1 ID, then you are not “on” any branch, but rather in “detached HEAD” mode, which happens when you check out some earlier commit to examine.

**$ git symbolic-ref HEAD**

refs/heads/master

# Check out a tagged commit, not at a branch tip.

**$ git checkout mytag**

**$ git symbolic-ref HEAD**

fatal: ref HEAD is not a symbolic ref

**$ git rev-parse HEAD**

1c7ed724236402d7426606b03ee38f34c662be27

# ... which matches the commit referred to by the

# tag.

**$ git rev-parse mytag^{commit}**

1c7ed724236402d7426606b03ee38f34c662be27

The HEAD commit is also often referred to as the “current” commit. If you are on a branch, it may also be called the “last” or “tip” commit of the branch.

A branch evolves over time; thus, if you are on the branch *master* and make a commit, Git does the following:

1. Creates a new commit with your changes to the repository content

2. Makes the commit at the current tip of the *master* branch the parent of the new commit

3. Adds the new commit to the object store

4. Changes the *master* branch (specifically, the ref refs/heads/master) to point to the new commit

“Deleting” a branch means simply deleting the corresponding ref; it has no immediate effect on the object store. In particular, deleting a branch does not delete any commits.

What it may do, however, is make certain commits *uninteresting,* in that they are no longer on any branch (that is, no longer reachable in the commit graph from any branch tip or tag).

set of all commits that contributed content to the latest commit. You can generally get the effect of looking “only at the history of this branch”—even though that’s not really well defined —with git log --first-parent.

The Index

The index is an independent data structure, separate from both your working tree and from any commit. It is simply a list of file pathnames together with associated attributes, usually including the ID of a blob in the object database holding the data for a version of that file. You can see the current contents of the index with git ls-files:

**$ git ls-files --abbrev --stage**

100644 2830ea0b

The --stage option means to show just the index; git ls-files can show various combinations and subsets of the index and your working tree, generally.

The index is the implicit source of the content for a normal commit. When you use git commit (without supplying specific pathnames), you might think that it creates the new commit based on your working files. It does not; instead, it simply realizes the current index as a new tree object, and makes the new commit from that.

The index does not just contain changes to be made on the next commit; it *is* the next commit, a complete catalog of the files that will be included in the tree of the next commit (recall that each commit refers to a tree object that is a complete snapshot of the repository content). When you check out a branch, Git resets the index to match the tip commit of that branch; you then modify the index with commands such as git add/mv/rm to indicate changes to be part of the next commit.

git add does not just note in the index that a file has changed; it actually adds the current file content to the object database as a new blob, and updates the index entry for that file to refer to that blob.

CHAPTER 2

Getting Started

Basic Configuration

Git configuration is in *~/.gitconfig*; this is a plain-text file, which you can edit directly as well, if you like. Its format is called *INI style* (after a file extension commonly used for it,

though not by Git), and is divided into sections, like so:

The parameters have full names qualified by the section in which they appear using a dot;

for example, the parameters mentioned in this example are:

* user.name

You use these names when reading or setting parameters with git config, rather than editing the file yourself. To set a parameter with git config:

**$ git config --{local,global,system}** *parameter value*

If you give this command when your current directory is inside a Git repository, it implies --local, and it will change the configuration for that repository only, in the file *.git/config*. Otherwise, the default is --global, which applies to your overall personal Git configuration in *~/.gitconfig*

Git reads these three configurations, each if available, in the order system, global, then local. Settings made in a later configuration override those from an earlier one so that, for example, you can set your normal email address with --global but change it for commits made in a specific repository if you use a different address when corresponding about that work.

Personal Identification

**$ git config --global user.name "Richard E. Silverman"**

**$ git config --global user.email** [**res@oreilly.com**](mailto:res@oreilly.com)

Text Editor

The default editor varies by platform; on Unix, it is the ubiquitous *vi*. You can customize this with the environment variables GIT\_EDITOR, EDITOR, or VISUAL (the latter two are respected

by many other Unix programs as well), or by setting core.editor. For example (reflecting the author’s predilections):

**$ git config --global core.editor emacs**

Command Aliases

Git has its own internal alias system as well, which may be more convenient. This command:

**$ git config --global alias.cp cherry-pick**

defines git cp as an alias for git cherry-pick. An exclamation point means to pass the alias definition to the shell, letting you use more complex aliases; for example, this definition in *~/.gitconfig*:

setup = ! "git init; git add .; git commit"

defines an alias git setup, which sets up a new repository using the contents of the current directory.

Getting Help

You can get help with a Git command or feature using Git itself, for example:

**$ git help commit**

Creating a New, Empty Repository

The command:

**$ git init *directory***

creates the argument directory if needed, and a directory named *.git* inside it holding a new, empty Git repository. Aside from the repository itself in *.git*, that directory will hold the *working tree*: copies of the files and directories under version control that you will edit. The *.git* directory holds the files and data structures that form the repository itself, including the database of all historical revisions of all project files. Unlike CVS and (until recently) Subversion, there is no control directory in each directory of the working tree (*CVS* and *.svn*); there is just

the one *.git* directory at the top of the project tree.

The default with no argument is the current directory; that is, a simple git init creates a new *.git* in the current directory. git init is a safe command. It will not remove any existing files in the target directory, the usual pattern being that you are about to add those files to the new repository. It will also not damage an existing repository, even though it gives a somewhat heartstopping message about “reinitializing” if you do it; all this actually does is make some administrative updates, such as picking up new templates for “hook” scripts made available by the system

administrator (see “Git Hooks” on page 196).

Selected Options

--bare

Creates a “bare” repository; that is, one without an associated working tree. The internal repository files that would otherwise be inside *.git* are instead created in the target directory

**$ git log --pretty=oneline**

cb9c236f Begin Project Foo!

**$ git show-ref master**

cb9c236f refs/heads/master

git log shows the ID of the most recent

Ignoring Files

While you’re working on a project, you may have files in your working directory that you want Git to simply ignore.

CHAPTER 13

Miscellaneous

Chapiter 14:

Fix the Last Commit I Made

**$ git commit --amend**

Edit the Previous n Commits

**$ git rebase -i HEAD~*n***

Undo My Last n Commits

**$ git reset HEAD~*n***

This removes the last *n* commits of a linear history from the current branch, leaving the corresponding changes in your working files. You can add --hard to make the working tree reflect the

new branch tip, but beware: this will also discard any current uncommitted changes, which you will lose with no recourse

Reuse the Message from an Existing Commit?

**$ git commit --reset-author -C *rev***

Add --edit to edit the message before committing

Reapply an Existing Commit from Another Branch?

**$ git cherry-pick *rev***

List Files with Conflicts when Merging

git status shows these as part of its report, but to just list their names

**$ git diff --name-only --diff-filter=U**

Get a Summary of My Branches?

* List local branches: git branch
* List all branches: git branch –a
* Get a compact summary of local branches and status with respect to their upstream counterparts: git branch –vv
* Get detail about the remote as well: git remote show origin (or other named remote)

Get a Summary of My Working Tree and Index State?

**$ git status**

Add -sb for a more compact listing; see the “Short Format” section of *git-status(1)* on how to interpret this.

Stage All the Current Changes to My Working Files?

**$ git add -A**

This does git add for every changed, new, and deleted file in your working tree. Add --force to include normally ignored files; you might do this when adding a new release to a “vendor branch,” which tracks updates to other projects you obtain by means other than Git (e.g., tarballs).

Show the Changes to My Working Files?

git diff shows unstaged changes; add --stage to see staged changes instead. Add --name-only or --name-status for a more compact listing.

List the Files in a Specific Commit?

**$ git ls-tree -r --name-only *rev***

This listing is restricted to the current directory; add –-fulltree for a complete list

git show *rev* is easier that git diff *rev*~ *rev*, and shows the author, timestamp, commit ID, and message as well. Add -s to suppress the diff and just see the latter information; use --name-status or --stat to summarize the changes. It also works for merge commits, showing conflicts from the merge as with git

log --cc (see “Showing Diffs” on page 142). The default for rev is HEAD.

List All Remotes?

git remote does this; add -v to see the corresponding URLs configured for push and pull (ordinarily the same):

Change the URL for a Remote?

**$ git remote set-url *remote URL***

Remove Old Remote-Tracking Branches?

**$ git remote prune origin**

This removes tracking for remote branches that have been deleted upstream

Find Commits I Made but Lost?

Perhaps after editing history with git rebase -i or git reset, or deleting a branch:

**$ git log –g**

Show the Changes for Each Commit?

git log -p shows the complete patch for each commit it lists, while these options summarize the changes in different ways:

**$ git log --name-status**

**$ git log –stat**

Show the Committer as well as the Author?

**$ git log --format=fuller**