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

Importing an Existing Project

In more detail: git add . adds the current directory to the (initially empty) index; this includes files as well as directories and their contents, and so on, recursively. git commit then creates a new tree object capturing the current state of the index, as well as a commit object with your comment text, personal identification, the current time, and so on, pointing to that tree. It records these in the object database, and then finally sets the *master* branch to the new commit; that is, makes the ref refs/heads/ master point to the new commit ID:

**$ 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.

Generally speaking, anything that is automatically generated you probably don’t want tracked by Git, and you don’t want Git constantly including them in listings or complaining about them either.

Syntax of “Ignore Patterns”

# Ignore this specific file in a subdirectory.

conf/config.h

# Ignore this specific file in the current directory.

# (not “./”)

/super-cool-program

## Patterns without slashes apply everywhere in this

## directory and below.

# Ignore individual objects and object archives

# (\*.o and \*.a).

\*.[oa]

# Ignore shared objects...

\*.so

# ... but don't ignore this file, or my boyfriend

# will complain.

!my.so

# Ignore any directories named “temp,” but still

# notice regular files and symbolic links with

# that name.

temp/

CHAPTER 3

Making Commits

git commit <FILENAME>

Giving a specific filename to git commit works differently: it ignores the index, and commits just the changes to that file.

Adding Partial Changes

**$ git add -p**

You can also add only *some* of the changes you’ve made to a file,

using git add --patch (-p).

git add -p allows you to conveniently split the work up into separate commits.

Just running git add -p with no arguments will let you examine all files with unstaged changes

git add -p is actually a special case of git add –interactive (-i). The latter starts at a higher level, allowing you to view status, add untracked files, revert to the HEAD version, select files to patch, etc.; git add -p just jumps straight to the “patch” subcommand of git add -i.

Shortcuts

git add -u

Include all files in the current index; this includes changed and deleted files, but not new ones.

git add -A

Include all filenames in the index and in the working tree; this stages new files as well.

use git add -A to stage all changes, additions, and deletions necessary to commit the new version

Removing a File

**$ git rm *filename***

This does two things:

1. Deletes the file’s entry from the index, scheduling it for removal in the next commit

2. Deletes the working file as well, as with rm *filename*

Renaming a File

Renaming a file or moving a directory in Git is simple, using the

git mv command:

**$ git mv foo bar**

This is actually just a shortcut for renaming the working file outside Git, then using git add on the new name:

**$ mv foo bar**

**$ git add bar**

Git doesn’t have a “rename” function internally at all; as indicated, git mv is just a

shortcut. If you run git status after the first command earlier, you’ll see what you’d expect: Git shows *foo* as deleted, and the new file *bar* as untracked. If you do it after the git add, though,

you see just one annotation: renamed: foo -> bar. Git sees that the file for a particular index entry has been removed from disk, while a new entry has appeared with a different filename—but the *same object ID*, and hence the same contents. It can also consider renaming relative to a less strict notion of file equivalence

This approach is very simple, but it requires that you sometimes be aware of the mechanics. For example: because this analysis is expensive, it is turned off by default when examining history with

git log; you have to remember to enable it with -M if you want to see renaming. Also, if you edit a file substantially *and* rename it in a single commit, it may not show up as a rename at all; you’re

better off editing, committing, then doing the rename in a separate commit to make sure it shows up as such

Unstaging Changes

just use git reset. This resets the index to match the current commit, undoing any changes you’ve made with git add. git reset reports the files with outstanding changes after its action:

With git reset --patch you can be even more specific, interactively selecting portions of your staged changes to unstage; it is the reverse of git add -p.

Making a Commit

Use git status first to check the files involved, and git diff --cached to check the actual changes

you’re applying. git diff alone shows any remaining *unstaged* changes (the difference between your working tree and the index);adding --cached (or the synonym --staged) shows the difference between the index and the last commit instead (i.e., the changes you’re about to make with this commit).

CHAPTER 4

Undoing and Editing Commits

Changing the Last Commit

There’sno preparatory step; just make whatever corrections you need,

adding these to the index as usual. Then use this command

**$ git commit --amend**

you have no way to refer to it, even though it’s still in the object database. Git has a feature to save you, though, called the *reflog:*

**$ git log -g**

The git log command, which we will discuss in Chapter 9, normally shows the history of your project via portions of the commit graph.

The -g option shows something entirely different, however. For each branch, Git maintains a log of operations performed while on that branch, called its “reflog.” Recall that a branch is just a ref pointing to the tip commit of the branch; each ref can have a log recording its referents over time. git log –g displays a composite reflog, starting with the current branch and chaining back through commands that switch branches, such as git checkout.

Discarding the Last Commit

**$ git reset HEAD~**

git reset is a versatile command, with several modes and actions. It always moves the head of the current branch to a given commit, but differs in how it treats the working tree and index; in this usage, it updates the index but leaves the working tree alone.

--mixed

The default: makes the index match the given commit, but does not change the working files. Changes made since the last commit appear unstaged.

--soft

This resets the branch tip only, and does not change the index; the discarded commit’s changes remain staged. You might use this to stage all the changes from several previous commits, and then reapply them as a single commit.

--hard

Resets your working files to match the given commit, as well as the index. Any changes you’ve made since the discarded commit are permanently lost, so be careful with this option! Resist the urge to make an alias or shortcut for using git reset --hard; you will probably regret it.

Undoing a Commit

Make a new commit undoing the earlier commit’s changes. The command git revert makes this easy; just give it the commit you want to undo:

**$ git revert 9c6a1fad**

Partial Undo

If you only want to undo some of the changes from an earlier commit, you can use a combination of commands we’ve seen before:

**$ git revert -n *commit***

**$ git reset**

**$ git add -p**

**$ git commit**

**$ git checkout**

The -n option to git revert tells Git to apply and stage the reverted changes, but stop short of making a commit. You then unstage all the changes with git reset, and restage only those you want using the interactive git add -p. Finally, after committing the subset of changes you want, you discard the rest by checking out the contents of the index, overwriting the remaining applied changes from git revert.

Plain git revert will complain if you have staged changes in the index (that is, the index does not match the HEAD commit), since its purpose is to make a new commit based on the one to be reverted, and it would lose your changes if it reset the index in order to do that. git revert -n, though, will *not* complain about that, since it is *not* making a commit.

Note that if the commit you’re reverting deleted a file, then this will add it back. After git reset though, the recovered file willappear as “untracked” to Git, and git add -p will not see it; you’ll

have to add it again separately, if it’s one of the changes you want to make (git add --interactive (-i) can help with that; it’s more general, and git add -p is actually a commonly used subcommand

of it). Similarly, the final checkout will not remove a restored file that you chose not to add; you’ll have to remove it yourself. You can use git reset --hard or git clean, but be careful not to accidentally remove other untracked files or revert other working tree changes you may have.

Editing a Series of Commits

This command:

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

rewrites the last *n* commits on the current branch

pick:

Use the commit as-is. Git will not stop for this commit unless there is a conflict

reword

Change just the commit message. Git allows you to edit the message before reapplying this commit.

edit

Change the commit contents (and message, if you want). Here, Git stops after remaking this commit and allows you to do whatever you want. The usual thing is to use git commit --amend to replace the commit, then git rebase --continue to let Git continue with the rebase operation.

However, you could also insert further commits, perhaps splitting the original changes up into several smaller commits. Git simply picks up from where you leave off, with the next change you asked it to make.

squash

Make this commit’s changes part of the preceding one. To meld several consecutive commits into one, leave the first one marked pick and mark the remaining ones with squash. Git concatenates all the commit messages for you to edit.

Fixup

Like squash, but discard the message of this commit when composing the composite message.

You can abbreviate an action to just its initial letter, such as r for reword. You can also reorder the lines to make the new commits in a different order, or remove a commit entirely by deleting its

line. If you want to cancel the rebase, just save a file with no action lines; Git will abort if it finds nothing to do. It will *not* abort if you just leave the directions as you found them, but the result will be the same in this simple case, since Git will find it does not need to remake any commits in order to follow the directions (which say to use each commit as-is with pick). At any point when Git stops, you can abort the entire process and return to your previous state with git rebase --abort.

Conflicts

It’s possible to ask for changes that invalidate the existing commits.For example: if one commit adds a file and a later commit changes that file, and you reverse the order of these commits, then Git cannot apply the new first patch, since it says to alter a file that doesn’t yet exist. Also, patches to existing files rely on context, which may change if you edit the contents of earlier

commits. In this case, Git will stop, indicate the problem, and ask you to resolve the conflict before proceeding.

CHAPTER 5

Branching

Switching Branches

**$ git symbolic-ref HEAD**

refs/heads/theodore

**$ git checkout simon**

Switched to branch 'simon'

**$ git symbolic-ref HEAD**

refs/heads/simon

Here, git symbolic-ref HEAD shows the ref (branch name) to which HEAD points:

This attempts to do three things:

1. Change the HEAD symref to point to the *commander* branch

2. Reset the index to match the tip of the new branch

3. Update the working tree index to match the index (this is called “checking out” the index, which gives the commandits name)

If these succeed, then you are now on the *commander* branch, with an index and working tree that match the tip of that branch. The following are some possible complications.

Uncommitted Changes

Suppose you have uncommitted changes to a tracked file when you try to switch branches. There are now four versions of the file in play: the two in the tip commits of the *master* and *commander* branches, and the two in your working tree and index (one or both of which have been altered, depending on whether you have staged the changes with git add). If the committed versions in the current and destination branches are the same, then Git will preserve your altered versions when switching branches, since they represent the same sets of changes in the new branch as in the old. It reminds you of a modified file *foo* thus:

**$ git checkout commander**

M foo

If the committed versions differ, however, or if the file does not exist at all in the destination branch, then Git warns you and refuses to switch:

Check Out with Merge

git checkout has a --merge (-m) option to help with this case. It performs a three-way merge between your working tree and the new branch, with the current branch as the base; it leaves you on the new branch, with the merge result in the working tree. As with any merge, you may have conflicts to resolve

If *simon* is fully merged in the current branch, then Git deletes it with no complaint. If it is not, but it is fully merged in its upstream branch, then Git proceeds with a warning:

**$ git branch -d simon**

Since Git doesn’t check other branches, it may be safe to delete a

branch because you know it is fully merged into another one; you can do this with the -D option as indicated, or switch to that branch first and let Git confirm the fully merged status for you.

Deleting the branch from the origin repository is not so obvious:

**$ git push origin :simon**

This is the general syntax for directly updating a remote ref. In this case, the local object name to the left of the colon is blank, meaning to just delete the remote ref.

Renaming a Branch

Renaming a local branch is simple:

**$ git branch -m *old new***

There is no direct way to rename the corresponding branch in a remote repository, however; you must separately push the new branch and delete the old one:

**$ git push -u origin *new***

**$ git push origin :*old***

You will need to tell others that you’ve done this, since when they pull they will get the new branch, but they will have to manually delete the old name with git branch -d. “Renaming” a branch is not actually a Git operation per se; git branch -m is just a shortcut for the create/delete routine.

CHAPTER 6

Tracking Other Repositories

When you check out a branch that doesn’t yet exist, but there is a remote-tracking branch by that name, Git automatically creates it and sets its upstream to be that tracking branch, so that subsequent push/pull operations will synchronize your local version of this branch with the remote’s version. For example, when you first clone a repository, Git checks out the remote’s HEAD branch, so this happens right away for one branch

Pushing

If you have added a local branch of your own and want to start sharing it with others, use the -u option to have Git add your branch to the remote, and set up tracking for your local branch in the usual way, for example:

**$ git push -u origin new-branch**

After this initial setup you can use just git push on this branch, with no options or arguments, to push to the same remote.

Push Defaults

There are several approaches Git can use when given no specific remote and ref to push (just plain git push, as opposed to git push *remote branch*):

* matching

Push all branches with matching local and remote names

* upstream

Push the current branch to its upstream (making push and pull symmetric operations)

* simple

Like upstream, but check that the branch names are the same (to guard against mistaken upstream settings)

* current

Push the current branch to a remote one with the same name (creating it if necessary)

* nothing

Push nothing (require explicit arguments)

The default as of this writing is matching, but with Git 2.0, this will change to simple, which is more conservative and avoids easy accidental pushing of changes on other branches that are not yet ready to be published.

The command git remote show *remote* gives a useful summary of the status of your repository in relation to a remote

**$ git remote show origin**

Note that unlike most informational commands, this actually examines the remote repository, so it will run *ssh* or otherwise use the network if necessary. You can use the –n switch to avoid this; Git will skip those operations that require contacting the remote and note them as such in the output.

git branch -vv gives a more compact summary without contacting the remote (and thus reflects the state as of the last fetch or pull; remember that the remote might have changed in the meantime

CHAPTER 7

Merging

You have changed the same two lines on each side in different ways, and Git’s line-oriented merge approach will not attempt to guess at your intent or combine the lines (e.g., form a single line

dolphin monoliths, interesting as those might be); it will signal a merge conflict:

“Unmerged paths” are files with conflicts Git could not resolve. To find out what went wrong in detail, use git diff. This command not only shows the differences between various combinations of working tree, index, and commits; it also has a special mode for helping with merge conflicts:

As usual, git diff shows differences between the working tree and the index, which in this case are the conflicts yet to be resolved; changes already made and staged are not shown. You can use git diff --staged to see those; add --stat for an overview.

In simple cases, you may get what you think of as the history

of “this branch” with git log --first-parent, which just

follows the first parent of merge commits rather than all of

them.

Resolving Merge Conflicts

Git doesn’t have built-in tools to interactively address merge conflicts directly; that’s what external merge tools are for; However, here are some tips for use in simple cases.

* git log -p --merge shows all commits containing changes relevant to any unmerged files, on either branch, together with their diffs.
* If you want to discard all the changes from one side of the merge, use git checkout --{ours,theirs} *file* to update the working file with the copy from the current or other branch, followed by git add *file* to stage the change and mark the conflict as resolved.
* Having done that, if you would like to apply *some* of the changes from the opposite side, use git checkout –p *branch file*. This starts an interactive loop that allows you to selectively apply or edit differing sections

In our example, if you decided to keep your version as a default, but selectively apply changes from the other branch, you could do:

**$ git checkout --ours moebius**

**$ git add moebius**

**$ git checkout -p floyd moebius**

**$ git add moebius**

Notes

* If the current branch is already contained in the other (that is, HEAD is an ancestor of the other branch tip), then git merge will just move the current branch up to meet the other in a “fast-forward” update, and not make a new commit at all. You can force a merge commit anyway with git merge --no-ff (“no fast-forward”), if you have some reason to do so.
* If you want to use Git’s content-merging and conflictresolution machinery, but do not want to create a merge commit, use git merge --squash.
* You can use git merge -m to specify a commit message just as with git commit
* Use git merge --no-commit to stop Git from committing when an automatic merge succeeds, in case you want to have a look first. This isn’t strictly necessary, since you could always abort the commit by giving a blank commit message, or make any changes you want afterward and use git commit --amend.
* Git records that a merge is in progress by setting the ref MERGE\_HEAD to point to the other branch; this is how it knows to make a merge commit

Details on Merging

Git finds automatically. Since our merge now involves three commits—two branch tips and the merge base—it is called a “three-way merge.

When there is a merge conflict for a file, Git simply stores not one version of the file in the index, but three:those belonging to the merge base and to the current and “other” branches, numbered 1, 2, and 3, respectively. The number is called the “stage” of the file and is a distinct property of an index entry along with the filename, mode bits, and so on. In fact, there is a third stage, 0, which is the usual state of an entry that has no associated merge conflict. We can see this using git ls-files,which shows the contents of the index. Prior to the merge, we see:

Prior to the merge, we see:

**$ git ls-files -s --abbrev**

100644 1fcbe134 0 moebius

After running git merge floyd and getting a conflict for this file, we see something very different (using -u instead of -s would show only unmerged paths; here we have only one file anyway):

**$ git ls-files -s --abbrev**

100644 30b7cdab 1 moebius

100644 1fcbe134 2 moebius

100644 08dbe186 3 moebius

You can use git cat-file to see the contents of the different stages, here the stage 1 merge-base version:

**$ git cat-file -p 30b7cdab**

You can refer to a specific stage of a file with the syntax :*n*:path; so git show :1:moebius is an easier equivalent for this.

Git records the three commits into the index in this way at the start of the merge. It then follows a set of simple rules to quickly resolve the easy cases; for example

* If all three stages match, reduce to a single stage 0.
* If stage 1 matches stage 2, then reduce to a single stage 0 matching stage 3 (or vice versa): one side made a change while the other did nothing
* If stage 1 matches stage 2, but there is no stage 3, then remove the file: we made no change, while the other branch deleted it, so accept the other branch’s deletion.
* If stages 1 and 2 differ, and there is no stage 3, then report a “modify/delete” conflict: we changed the file, while the other branch deleted it; the user must decide what to do.

Merge Tools

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**