Git Pocket Guide

Contents

[CHAPTER 2 Getting Started 5](#_Toc157260066)

[CHAPTER 3 Making Commits 6](#_Toc157260067)

[CHAPTER 4 Undoing and Editing Commits 7](#_Toc157260068)

[CHAPTER 7 Merging 10](#_Toc157260069)

[CHAPTER 8 Naming Commits 12](#_Toc157260070)

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.

An implication of this behavior that occasionally confuses people is that if you change a file, git add it, then change it again, it is the version you last added to the index, *not* the one in your working tree, that is part of the next commit. git status shows this explicitly, by listing the same file as having both “changes to be committed” and “changes not staged for commit.”

Similar to git commit, git diff without arguments also has the index as an implicit operand; it shows the differences between your working tree and the *index*, rather than the current commit

The idea is that git diff shows changes not yet staged for commit, so you can see what you have yet to deal with (or have deliberately not included) as you prepare the next commit. git diff --staged shows the opposite: the differences between the index and the current commit (that is, the changes that are about to be committed).

Merging

There are two aspects to merging in Git: content and history.

Merging Content

Merging History

Push and Pull

When you clone a repository, such as with git clone server:dir/repo, here’s what Git does:

1. Creates a new repository.

2. Adds a *remote* named “origin” to refer to the repository being cloned in *.git/config*:

[remote "origin"]

fetch = +refs/heads/\*:refs/remotes/origin/\*

url = server:dir/repo

The fetch value here, called a *refspec*, specifies a correspondence between sets of refs in the two repositories: the pattern on the left side of the colon names refs in the remote, and the spec indicates with the pattern on the right side where the corresponding refs should appear in the local

repository. In this case, it means: “Keep copies of the branch refs of the remote origin in its local namespace in this repository, refs/remotes/origin/.”

3. Runs git fetch origin, which updates our local refs for the remote’s branches (creating them in this case), and asks the remote to send any objects we need to complete the history for those refs (in the case of this new repository, all of them).

4. Finally, Git checks out the remote’s current branch (its HEAD ref), leaving you with a working tree to look at. You can select a different initial branch to check out with --branch, or suppress the checkout entirely with -n.

So, Git created a new branch with the same name in your local namespace, starting at the same

commit as the remote branch:

**$ git show-ref --abbrev master**

d2e46a81 refs/heads/master

d2e46a81 refs/remotes/origin/master

Finally, Git has a special convenience for git checkout if you try to check out a branch that doesn’t exist, but a corresponding branch *does* exist as part of a remote. It will automatically set up

a local branch by the same name with the upstream configuration just demonstrated. For example:

**$ git checkout beta**

git pull

Runs git fetch on the remote for the current branch, updating the remote’s local tracking refs and obtaining any new objects needed to complete the history of those refs: that is,

all commits, tags, trees, and blobs reachable from the new branch tips. Then it tries to update the current local branch to match the corresponding branch in the remote. If only one side has added content to the branch, then this will succeed, and is called a *fast-forward* update since one ref is simply moved forward along the branch to catch up with the other. If both sides have committed to the branch, however, then Git has to do something to incorporate both versions of the branch history into one shared version. By default, this is a merge: Git merges the remote branch into the local one, producing a new commit that refers to both sides of the history via its parent pointers. Another possibility is to *rebase* instead, which attempts to rewrite your divergent commits as new ones at the tip of the updated remote branch (see

“Pull with Rebase” on page 89).

git push

Attempts to update the corresponding branch in the remote with your local state, sending any objects the remote needs to complete the new history. This will fail if the update would be non–fast-forward as described earlier (i.e., would cause the remote to discard history), and Git will suggest that you first pull in order to resolve the discrepancies and produce an acceptable update.

# CHAPTER 2 Getting Started

Getting Help

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

**$ git help commit**

**$ 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

# 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 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

The -y switch to git mergetool tells it to run the tool on all unmerged files, without pausing to prompt yes or no for each one.

git mergetool leaves a backup *foo.orig* for each file *foo* it processes, since you might have modified it yourself before running the merge tool. You can set mergetool.keepBackup no to turn off this feature. Actually, Git still makes the backup; it just deletes it when the merge tool exits successfully, so that the backup is still there in case the tool were to crash.

Merge Strategies

git merge -s ours

* The ours strategy is simple: it discards all changes from the other branch. This leaves the content on your branch unchanged, and when you next merge from the other branch, Git will only consider changes made from this point forward. You might use this to retain the history of a branch, without incorporating its effects.

git merge -s recursive -X ours

This is the ours option to the recursive strategy, not to be confused with the ours strategy. The recursive strategy is often the default, and so you might not have to use -s, but we’ll be explicit here. This option directs Git to resolve conflicting changes in favor of the current branch. This is different from the ours strategy, in that nonconflicting changes can still be resolved in favor of either branch. You can use -X theirs as well, to resolve in favor of the other branch instead.

ignore-space-change*,* ignore-all-space*,* ignore-space-at-eol

These options to the recursive strategy automatically resolve conflicts differing only in certain types of whitespace; see *git-merge(1)* for details.

# CHAPTER 8 Naming Commits

The command **git rev-parse** is useful for checking your understanding: it will take a name in the various syntaxes presented here and translate it to an object ID, so you can make sure it refers to the object you expected. For names that represent sets of commits, git rev-list will show the resulting set.

**git describe**

The output of the git describe command, which names commits relative to a tag; for example, *v1.7.12-146-g16d26b16* refers to commit *16d26b16*, which is *146* commits away from the tag v1.7.12. As output, this might be used as part of a build identifier, where it suggests to the reader the proximity of the build to a tag with a presumably helpful name. As input to Git however, only the trailing hex digits after -g are meaningful, and are used as an abbreviated commit ID.

Ref Name

A simple ref points directly to an object ID. Git follows a symbolic ref such as “master” until it finds a simple ref; for example, HEAD points to *master* if you are that branch, and *master* points to the commit at the branch tip. If the object is a tag rather than a commit, then Git follows the tag (possibly through intermediate tags) until it reaches a commit.

There are several rules for expanding ref names, allowing you to use short names in most situations rather than fully qualified names such as refs/heads/master. To find a ref named foo, Git looks for the following in order:

* foo: Normally, these are refs used by Git internally, such as HEAD, MERGE\_HEAD, FETCH\_HEAD, and so on, and are represented as files directly under *.git*

\* refs/foo

\* refs/tags/foo: The namespace for tags

\* refs/heads/foo: The namespace for local branches

\* refs/remotes/foo: The namespace for remotes, though this would not ordinarily itself be a ref, but rather a directory containing the remote’s refs

\* refs/remotes/foo/HEAD: The default branch of the remote“foo”

Briefly, this means that git checkout foo will check out a tag named *foo* if there is one, otherwise, a branch; if there is neither, but there is a remote named *foo*, then it will check out the default branch of that remote.

Names Relative to a Given Commit:

*rev*^*n*

This refers to the *n*th parent of a commit, numbered starting at 1.

*rev*^ = *rev*^1

*rev*^0 = rev if rev is a commit. If rev is a tag, then *rev*^0 is the commit to which the tag refers

In a linear history, *rev*^ is the previous commit to rev, and *rev*^^ the commit two steps back

*rev*^^ ≠ *rev*^2.



*rev~n*

this is the *n*th ancestor of rev, always following the first parent commit. Special cases

*rev*~ = *rev*~1

*rev*~0 = rev

HEAD~2 = HEAD^1^1 = HEAD^^,

Names Relative to the Reflog

Local branch names usually have a reflog: a log of commits that

used to be the head of this branch, along with the actions that

changed it each time: commit, cherry-pick, reset, and so on. You

view the composite log with git log -g, which follows your trail

from one branch log to another via checkouts. The syntax

refname@\{selector\} allows you to name a single commit according

to various criteria evaluated against your reflog:

*refname*@{*time/date*}

The commit named by this ref at the given point in time. The time can be specified in a very flexible format

* Now
* Yesterday
* last week
* 6 months ago
* two Saturdays past
* Sat Sep 8 02:09:07 2012 -0400 (or meaningful subsets of this)
* 1966-12-06 04:33:00

*refname*@{*n*}

For nonnegative *n*, this is the *n*th prior value of refname (zero refers to the current value and is a synonym for refname).Note that this need *not* be the same as *refname*~*n*, the *n*th prior commit on the branch!

You can omit refname to refer to the current branch (e.g., @{5}).

@{*-n*}

With a negative number, this is the current tip of the *n*th branch checked out before the current one. For example, if you’re on *master* and switch to *foo* with git checkout foo, then git checkout @{-1} will take you back to *master*

The Upstream Branch

The notation foo@{upstream} (or just foo@{u}) names the branch upstream of the branch *foo*, as defined by the repository configuration. This is usually arranged automatically when checking out a local branch corresponding to a remote one, but may be set explicitly with commands such as git checkout --track, git branch --set-upstream-to, and git push -u. It just gives the object ID of the upstream branch head, though; options to git rev-parse are useful to find out the upstream branch name:

**$ git rev-parse HEAD@{upstream}**

**$ git rev-parse --abbrev-ref HEAD@{upstream}**

**$ git rev-parse --symbolic-full-name HEAD@{upstream}**

Following Chains

There are various kinds of pointers or indirection in Git: a tag points to another object (usually a commit); a commit points to the tree representing the content of that commit; a tree points to

its subtrees; and so on. The syntax *rev^type* tells Git to recursively dereference the object named by rev until it reaches an object of the given type. For example:

* release-4.1^{commit} names the commit tagged by release-4.1, even if there are intermediate tags.
* master~3^{tree} names the tree associated with the third commit back from the tip of the *master* branch.

You don’t often have to use these kinds of names, as Git is smart about doing this automatically when appropriate. If you give a tag to git checkout, it knows you mean to check out the tagged

commit; similarly, if you want to list the filenames in a commit, git ls-tree -r master~3 would be sufficient. However, sometimes you need to be more precise: git show release-4.1 would show both the tag and the commit; you could use release-4.1^{commit} to show only the commit. Special cases:

• *rev*^0 is a synonym for *rev*^{commit}.

• *rev*^{} means to follow the chain to the first nontag object (of whatever type).

Addressing Pathnames

The notation *rev*:*path* names a file by pathname in a given commit

(e.g., olympus@{last.week}:pantheon/zeus). Actually, it’s

more general than that: recall from “The Object Store” on page

6 that a pathname *foo/bar/baz* names an object in some tree, either

a blob (the contents of a file *baz*) or another tree (the entries

in a directory *baz*). So rev can be any tree-like object: a tree (obviously),

a commit (which has an associated tree), or the index, and the object selected by path may be a blob (file) or another tree (directory). Special cases:

:*path*

Addresses an object in the index.

:*n*:*path*

Addresses an object in the index, including its stage number (see “Details on Merging” on page 105); :*path* is actually short for :0:*path*.

Naming Sets of Commits

A

Add all commits reachable from A.

^A

Remove all commits reachable from A.

A^@

Add all commits reachable from A, but exclude A itself. This acts like a macro that expands to the list of parents of A, which are then interpreted according to (1).

Here are some useful abbreviations

A..B *=* ^A B

This is all commits reachable from B but not from A. Note that this excludes A itself.

A...B *=* A B --not $(git merge-base A B)

This is all commits reachable from either A or B, but not from

both. It is called the *symmetric difference*, which is the name

for the corresponding set operation: (A ∪ B) − (A ∩ B).

For the .. and ... operators, a missing commit name on either side defaults to HEAD

WARNING

The use of this set notation depends on context. git log interprets its arguments just as shown in this section, indicating the set of commits on which it should report. Git checkout, however, does not accept it, since it doesn’t make sense to check out more than one commit at a time. And git show treats individual revs as naming just one commit (rather than all commits reachable from it), but accepts compound forms such as A..B. Note too that git diff also uses the .. and ... syntaxes with pairs of commits—but with entirely different meanings! Git diff A..B is just a synonym for git diff A B. Caveat Gittor.

CHAPTER 9

Viewing History

The primary command for examining the commit history of your repository is git log. The documentation for this command, *git-log(1)*, is about 30 pages long

The format of the command is:

**$ git log [*options*] [*commits*] [[--] *path* ...]**

git log

The default for commits is HEAD, so this lists the commits reachable from the current HEAD commit. This is generally a branch, but may not be if you have checked out an arbitrary commit and are in “detached HEAD” mode

git log topic

Lists the commits in the *topic* branch, even if you are on another branch.

git log alvin simon

Lists all commits on either of the branches *alvin* or *simon*.

git log alvin..simon

Lists all commits in *simon* that are not in *alvin*; this is often those commits on *simon* that have occurred since you last merged it with *alvin*.

Output Formats

The default output format is fairly detailed, including the author timestamp and commit message:

git log --oneline gives more compact output, including just the ID and message subject for each commit The --oneline option is actually short for --format=oneline --abbrev-commit, and the default is --format=medium. There are a number of predefined formats; the following table shows the full list, along with some commit elements they contain (they all show the commit ID).



Defining Your Own Formats

You can also customize the display format, with git log –format ="format:*string*". You can give a format using a set of substitutions similar in usage to the printf function in the C standard library (and widely copied in other languages).

*# committer, commit ID, relative timestamp, subject*

**$ git log --date=relative --format='%an, %h, %ar, "%s"**

This example uses color and underlining to distinguish the different fields on the line. The colors may not show here depending on the medium in which you’re reading this text, but give it a try (it assumes your terminal is set up to handle color, of course):

*# commit ID, subject, committer, date*

**$ git log --date=short --format=\**

**"%C(blue)%h %C(reset)%s %C(magenta)%aN %C(green ul)\**

**%ad%C(reset)"**

Make sure to use %Creset at the end of such a format; otherwise, if the output is going directly to a terminal rather than through a pager, you’ll leave the terminal stuck in whatever color or mode you last used

Notes

* --pretty is a synonym for --format (from the term “pretty printing”).

Limiting Commits to Be Shown

There are many options for further limiting the commits to be shown beyond the *commits* expression given as an argument to git log; here is a selection of common ones:

-*n* (-n *n*, --max-count=*n*)

Only show the first *n* commits.

--{before,after}=*date*

Show commits made before or after a specific date (synonyms: --{until,since}). Note that this refers to the commit timestamp; there is no analogous simple way to refer to the

author timestamp.

--{author,committer}=*regexp*

Show only commits whose author or committer header (name <email>) matches the given regular expression. Multiple instances of a given constraint are combined with logical“or,” but (as usual) use of both types counts as logical “and”; thus, git log --author=Richard --author=Booboo --committer=Felix shows commits made by Felix, whose author is either Richard or Booboo.

--grep=*regexp*

Show only commits whose log messages match the given regular expression. Multiple instances are combined with logical “or”; change this to “and” with --all-match. Use --grep-reflog to match reflog entries instead, when using git log -g to examine the reflog instead of the commit graph (--grep still matches the commit message, even though the commits examined are found via the reflog; it does not match the reflog comment instead).

--first-parent

Follow only the first parent of a merge commit, rather than all of them. This can give a more useful history of a topic branch into which you periodically merge from a more central branch, keeping it up to date with the main development.This shows only the activity on the topic branch itself, rather than commits brought in from the main branch by merging.

Regular Expressions

A number of options affect the interpretation of regular expressions:

-i (--regexp-ignore-case)

Ignore case differences (e.g., hello and HELLO will both match “Hello”).

-E (--extended-regexp)

Use extended regular expressions; the default type is basic.

-F (--fixed-strings)

Consider the limiting patterns as literal strings to be matched; that is, don’t interpret them as regular expressions at all.

Reflog

git log --walk-reflog (-g) shows a completely different log: the *reflog*. This is a log of actions you’ve taken in your repository,and it can be very helpful in recovering from mistakes;

Decoration

git log --decorate={no,short,full} shows refs pointing to the listed commits

Date Style

git log --date= {local,relative,default,iso,rfc,short,raw}

This option affects how dates are rendered in formatted log

output, as long as the format has not explicitly given a date

style. For example, using this format:

[pretty]

compact = %h %ad, \"%s\"

**$ git log -1 --format=compact --date=local**

Listing Changed Files

git log --name-status summarizes which files changed in a given commit (relative to its predecessor), and the nature of the changes:

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

git log --name-only lists only filenames without the status codes, and --stat gives an ASCII-art graph (“diffstat”) representing the amount and kind of change in each file:

**$ git log --stat**

git log --dirstat summarizes the amount of change in subdirectories (it can take a number of parameters controlling how the summarization is done):

**$ git log --dirstat**

Showing and Following Renames or Copies

Ordinary git log does not show file renaming, because it takes longer to do this and often you’re not interested. To enable renaming detection, use --find-renames[=*n*] (-M[*n*]). The optional integer *n* is an index of similarity: consider a delete/add pair to be a rename if the before/after files are at least *n*% identical (the default is 100%):

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

To have Git follow a file past a rename, use git log --follow; this only works when you give a single file to follow

**$ git log --follow bar**

Searching for Changes: The “pickaxe”

The Git “pickaxe,” git log -S *string*, lists commits that changed the number of occurrences of string in at least one file.

For example, if you want to know when a particular feature was added, using this command with the name of a function or variable specific to the feature will turn it up, as the earliest commit that introduced that term. git log -G *pattern* does the same with a regular expression.

If you combine the pickaxe with a git log option that lists files, such as --name-status, Git shows only those files that triggered the listing (those in which the number of string or pattern occurrences changed). If you add --pickaxe-all, then Git shows *all* files touched by the listed commits. This allows you to see the entire changeset associated with any commit that matched the pattern you’re interested in.

Showing Diffs

git log -p shows the “patch” or “diff ” associated with each commit (illustrating the actual changes made to the files, only for text files, naturally), after the usual commit information as indicated by the log format in use

Normally, no diff is shown for merge commits, however you can use these options:

-m:

Shows each pairwise diff between the merge and its parents

-c

Shows the differences with all parents simultaneously in a merged format (a generalization of

the traditional “unified diff ”), rather than serially as with -m, and only for files that

were modified in all branches.

Word Diff

The option --word-diff[={plain,color,none}] shows wordlevel changes within lines, rather than entire changed lines

Comparing Branches



git log --cherry-pick takes this into account by omitting commits that have identical diffs

**$ git log master...other**

e5feb479 E

070e87e5 D

9b0e3dc5 C

6f70a016 3

0badfe94 2

15f47204 1

**$ git log --cherry-pick master...other**

e5feb479 E

9b0e3dc5 C

6f70a016 3

15f47204 1

The variation --cherry-mark will mark duplicate commits with an equal sign, instead of omitting them:

**$ git log --cherry-mark master...other**

+ e5feb479 E

= 070e87e5 D

+ 9b0e3dc5 C

+ 6f70a016 3

= 0badfe94 2

+ 15f47204 1

Displaying Sides

git log master..other (with just two dots) shows one side of this situation: those commits on *other* that are not on *master*

If

you want cherry-pick detection, you have to consider both sides as before, but then you are no longer viewing just one side. You can recover this by adding --{left,right}-only:

**$ git log master..other**

6f70a016 3

0badfe94 2

15f47204 1

**$ git log --cherry-pick --right-only master...other**

6f70a016 3

15f47204 1

This shows commits on *other* that are not contained in *master* or patch-equivalent to another commit in their difference, in this case, omitting commit *2* since it is equivalent to *D*. And similar to --cherry-mark, the related option --left-right displays the side of a commit with the symbols < and >:

**$ git log --cherry-mark --left-right master...other**

< e5feb479 E

< 070e87e5 D

= 9b0e3dc5 C

> 6f70a016 3

= 0badfe94 2

> 15f47204 1

The simple option --cherry is a synonym for --right-only --cherry-mark --no-merges, so that this: **$ git log --cherry HEAD@{upstream}...** shows the commits on your side of the current branch (ignoring possible merges with other branches), marking those that duplicate changes made by distinct commits on the other side (probable cherry-picks, either literally or by another means such as applying patches via email with git format-patch and git am).

Commit Ordering

• --reverse reverses the output list.

History Simplification

Related Commands

git cherry

This command is similar to git log --cherry, but more specialized. It shows commits on a branch that are not in the upstream, marking those whose changes are duplicated by distinct upstream commits with a minus sign (while other commits have a plus sign). Using the same example as before, if we’re on the *other* branch for which *master* is upstream:

**$ git cherry -v --abbrev**

+ 6f70a016 3

- 0badfe94 2

+ 15f47204 1

git shortlog

As mentioned earlier, git shortlog summarizes commit history, grouping commits by author with the number of commits and their subjects, and applying a mailmap if available to rewrite author

names or email addresses

This can be useful in preparing the release notes for a new version of a project, automatically collating the new features in this release. You could limit it to just the features since the last version by referring to the tags for the previous and current release (e.g., git shortlog v1.0..v1.1).

CHAPTER 10

Editing History

Rebasing

The general purpose of git rebase is to move a branch from one location to another

The most general form of the command is:

**$ git rebase [--onto *newbase*] [*upstream*] [*branch*]**

which means to replay the commit set upstream..branch starting at newbase. The defaults are:

* *upstream:* HEAD@{upstream}

The upstream of the current branch, if any

* *branch:*

HEAD

* *newbase:*

The upstream argument, whatever its default or usersupplied value is

For example, given the commit graph in Figure 10-1, the command git rebase --onto C master topic would move the *topic* branch as shown in Figure 10-2.



The behavior of the default arguments to git rebase reveals the simplest use of rebasing: keeping a sequence of local commits at the tip of a branch as the upstream progresses, rather than performing a merge.

Following the preceding defaults, the simple command

**$ git rebase**

actually means:

**$ git rebase --onto origin/master origin/master master**

The Big Hammer: git filter-branch

git filter-branch is the most general tool for altering the history of a repository. It walks the portion of the commit graph you specify (by default, the current branch), applying various filters

you supply and rewriting commits as necessary. You can use it to make wholesale programmatic alterations to the entire history. Since this is an advanced command, we will just sketch its operation and refer the reader to *git-filter-branch(1)* for more detail. You can apply the following filters, whose string arguments are passed to the shell. When they run, the environment contains the

following variables reflecting the commit being rewritten

CHAPTER 13

Miscellaneous

In this chapter, we cover some Git commands and topics that don’t fit easily into any of the foregoing discussions.

git cherry-pick

git cherry-pick allows you to apply the changeset of a given commit as a new commit on the current branch, preserving the original author information and commit message.

--edit (-e)

Edit the commit message before committing

--no-commit (-n)

Apply the patch to the working tree and index, but do not

commit. You can use this to take the commit’s changes as a

starting point for further work, or to squash the effect of

several cherry picked commits into a single one.

As with other commands that apply patches, git cherry-pick can fail if a patch does not apply cleanly, and it uses the merge machinery in that case, recording conflicts in the index and working files in the usual way. It then prompts you to use the options --{continue,quit,abort} to continue after resolving the conflicts, skip the current commit, or abort the whole cherry pick, similar to git rebase.

git grep

git grep lets you search your repository content using regular

expressions: not only the working tree, but also the index or any

commit in the history without having to check it out. You can

even use it outside a Git repository, as a more powerful version

of the usual Unix *grep* command

Combining Regular Expressions

Instead of a single regular expression, git grep can handle

Boolean combinations of expressions, combined with the options

--{and,or,not} in infix notation (“or” is the default connective;

“and” binds more tightly than “or”; use parentheses for

grouping, which you may have to escape to protect from your

shell). In this usage, patterns are preceded by -e. For example

**$ git grep -e '^#define' --and \( -e AGE\_MAX -e MAX\_AGE \)**

This finds lines that begin with #define and contain either AGE\_MAX or MAX\_AGE; thus, it finds both #define AGE\_MAX and #define MAX\_AGE.

What to Search

By default, git grep searches tracked files in the working tree, or

given commit or tree objects. The given objects must be listed

individually; you cannot use range expressions such as mas

ter..topic. You can add path limiters to restrict the files searched

to those matching at least one glob-style pattern.

What to Show

By default, git grep shows all matching lines, annotated with

filename and object as appropriate. Other options include

-n

Show line numbers

-h

Omit filenames

--count (-c)

Show the number of lines that match, rather than the matching

lines themselves

--files-with-matches (-l)

Just list the files containing matches

--files-without-matches (-L)

Just list the files containing no matches

--full-name

Show filenames relative to the working tree top, rather than

the current directory

--heading

Show the filename once before the matches in that file, rather

than on each line

How to Match

-i (--regexp-ignore-case)

Ignore case differences (e.g., hello and HELLO will both match “Hello”).

-E (--extended-regexp)

Use extended regular expressions; the default type is basic.

-F (--fixed-strings)

Consider the limiting patterns as literal strings to be matched;

that is, don’t interpret them as regular expressions at all.

git rev-parse

git rev-parse is a plumbing command, meant mainly for use by

other Git programs to parse and interpret portions of Git command

lines that use common options for specifying revisions

git clean

git clean removes untracked files from the working tree, optionally

limited by a glob pattern (e.g., git clean '\*~' to remove

backup files). Options include:

--force (-f)

Really do something. git clean will make no changes

without this flag, unless you set clean.requireForce to false.

--dry-run (-n)

Show what would be done, but remove no files.

--quiet (-q)

Report only errors, not the files removed.

--exclude=*pattern* (-e)

Add pattern to the “ignore” rules in effect.

-d

Remove untracked directories as well as files. Directories

that are in turn other Git repositories will not be removed

unless you add -f -f (two “force” flags).

-x

Skip the normal “ignore” rules (but still obey rules given with -e).

-X

Remove only ignored files.

git stash

git stash saves your current index and working tree, then resets

the working tree to match the HEAD commit as git reset --hard

would do

Subcommands

save

This is the default subcommand, saving the current working state

as described. Options include:

--keep-index

Do not revert changes already applied to the index.

--include-untracked (-u)

Save untracked files (normally only tracked files are saved).

This is useful to save compilation artifacts such as object

files, normally ignored and untracked but that would be

costly to recreate.

list

List the stack of stashes, which can be referred to symbolically as stash@{0}, stash@{1}, and so on (most recent first). You can add options as to git log.

show

Show the changes in a given stash, as the diff between the stash and its corresponding original worktree state. The default is the latest stash (stash@{0}), and you can add options as with git diff.

pop

The inverse of git stash: restore a stashed state and remove it from the stash list; the default state to use is stash@{0}, or you can supply a different stash. If the stash does not apply cleanly,

this does not remove the stash; use git stash drop after resolving the conflicts. With --index, restores the saved index as well(which is otherwise discarded).

apply

Like git stash pop, but does not remove the restored state from the stash list.

drop [stash]

Remove stash from the stash list (default stash@{0}).

clear

Deletes the entire stash list.

git show

git show displays a given object (default HEAD) in a manner

appropriate to the object type

For example, to see the diff from one commit to the next, you could use git diff foo~ foo, but git show foo is just simpler

git tag

A Git tag gives a stable, human-readable name to a commit, such as “version-1.0” or “release/2012-08-01”. There are two kinds of tags:

* A “lightweight tag” is just a ref in refs/tags pointing to the tagged commit.
* An “annotated tag” is also a ref in refs/tags, but pointing to a tag-type object instead, which in turn not only points to the tagged commit, but records other information as well: the tag author, timestamp, a tag message, and an optional GnuPG cryptographic signature.

git tag *tagname commit* creates a new lightweight tag pointing

to the given commit (default HEAD). Options include:

--annotate (-a)

Make an annotated tag instead

--force (-f)

Be willing to replace existing tags (this normally fails)

--delete (-d)

Delete a tag

--list *pattern* (-l)

List tags with names matching pattern. No pattern means

list all tags, and this is the default for a plain git tag command

without arguments. Multiple patterns means to list

tags matching at least one pattern.

--message="text" (-m)

Use text as the tag message (instead of invoking the editor).

Multiple -m options are concatenated as paragraphs. This

implies an annotated tag.

--file=*filename* (-F)

Use the contents of filename as the tag message (instead of

invoking the editor); “-” means standard input. This implies

an annotated tag.

Deleting a Tag from a Remote

Deleting a tag from your repository will not automatically delete

it from the origin when pushing; you have to do that explicitly:

**$ git push origin :*tagname***

Following Tags

When you pull (or fetch) from a configured remote, Git will automatically fetch new tags, but a “one-shot” pull specifying theremote repository (git pull *URL branch*) will not do this

For new tags you create, use git push --tags to send them when pushing.

git diff

git diff is a versatile command, showing the difference between content pairs in the working tree, commits, or index. The following are some common forms.

git diff

This shows your *unstaged* changes; that is, the difference between

the working tree and the index.

git diff --staged

This shows your *staged* changes; that is, the difference between

the latest commit and the index. These are the changes that will

be included in the next commit. --cached is a synonym for

--staged. You can give an alternate commit to compare as an

argument; the default is HEAD.

git diff <commit>

This shows the difference between the working tree and the

named commit

git diff <A> <B>

This shows the difference between two commits, trees, or blobs

A and B. A..B is a synonym for A B; note that this has no connection

to the meaning of that syntax when naming sets of commits

(see “Naming Sets of Commits” on page 123). If either A or

B is omitted in A..B, the default is HEAD; this syntax is thus useful

for specifying HEAD for one of these by just typing two dots,

which is easier and faster than typing in all caps.

You can limit the comparison to specific files with trailing patterns; for example, this shows the unstaged changes only in Java and C source files:

$ git diff -- '\*.java' '\*.[ch]'

git diff accepts quite a few options controlling how Git computes or displays differences, most of which it has in common with git log, which we discuss in Chapter 9. For example, this summarizes the differences instead of displaying them:

**$ git diff --stat**

foo.c | 1 +

icky.java | 1 +

3 files changed, 3 insertions(+)

and this just lists the files that contain differences:

**$ git diff --name-only**

foo.c

icky.java

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**