Contents

[1 Git Started 1](#_Toc527036538)

[2 Git Basics 3](#_Toc527036539)

[3 Git Branching 11](#_Toc527036540)

[4 Git on the Server 16](#_Toc527036541)

[5 Distributed Git 16](#_Toc527036542)

[6 GitHub 20](#_Toc527036543)

**Undoing Things**

If you want to redo that commit, make the additional changes you forgot, stage them, and commit again using the --amend option:

$ git commit –amend

**Unstaging a Staged File**

git reset HEAD <file> to unstagea file

**Unmodifying a Modified File**

$ git checkout -- CONTRIBUTING.md

It’s important to understand that git checkout -- <file> is a dangerous command. Any changes you made to that file are gone — Git just copied another file over it. Don’t ever use this command unless you absolutely know that you don’t want the file.

Remember, anything that is *committed* in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an --amend commit can be recovered (see Data Recovery for data recovery). However, anything you lose that was never committed is likely never to be seen again.

**Working with Remotes**

You can also specify -v, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

**Adding Remote Repositories**

To add a new remote Git repository as a shortname you can reference easily, run

$ git remote add <shortname> <url>:

**Fetching and Pulling from Your Remotes**

As you just saw, to get data from your remote projects, you can run:

$ git fetch <remote>

It’s important to note that the git fetch command only downloads the data to your local repository —

git pull command to automatically fetch and then merge that remote branch into your current branch

Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

**Pushing to Your Remotes**

The command for this is simple: git push <remote> <branch>. If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you’ve done back up to the server:

$ git push origin master

**Inspecting a Remote**

If you want to see more information about a particular remote, you can use the git remote show <remote> command

$ git remote show origin

This command shows which branch is automatically pushed to when you run git push while on certain branches. It also shows you which remote branches on the server you don’t yet have, which remote branches you have that have been removed from the server, and multiple local branches that are able to merge automatically with their remote-tracking branch when you run git pull.

**Renaming and Removing Remotes**

For instance, if you want to rename pb to paul, you can do so with git remote rename

If you want to remove a remote for some reason — you’ve moved the server or are no longer using a particular mirror, or perhaps a contributor isn’t contributing anymore — you can either use git remote remove or git remote rm:

**Tagging**

**Listing Your Tags**

Listing the available tags in Git is straightforward. Just type git tag (with optional -l or --list):

$ git tag

You can also search for tags that match a particular pattern

$ git tag -l "v1.8.5\*"

If you want just the entire list of tags, running the command git tag implicitly assumes you want a listing and provides one; the use of -l or --list in this case is optional. If, however, you’re supplying a wildcard pattern to match tag names, the use of –l or --list is mandatory.

**Creating Tags**

Git supports two types of tags: *lightweight* and *annotated*.

A lightweight tag is very much like a branch that doesn’t change — it’s just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. The easiest way is to specify -a when you run the tag command:

$ git tag -a v1.4 -m "my version 1.4"

The -m specifies a tagging message, which is stored with the tag. If you don’t specify a message for an annotated tag, Git launches your editor so you can type it in.

You can see the tag data along with the commit that was tagged by using the git show command

$ git show v1.4

**Lightweight Tags**

Another way to tag commits is with a lightweight tag. To create a lightweight tag, just provide a tag name:

$ git tag v1.4

**Tagging Later**

To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

$ git tag -a v1.2 9fceb02

**Sharing Tags**

By default, the git push command doesn’t transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches — you can run git push origin <tagname>.

$ git push origin v1.5

you can also use the --tags option to the git push command. This will transfer all of your tags to the remote server that are not already there.

$ git push origin –tags

**Checking out Tags**

If you want to view the versions of files a tag is pointing to, you can do a git checkout

$ git checkout 2.0.0

**Git Aliases**

If you don’t want to type the entire text of each of the Git commands, you can easily set up an alias for each command using git config

$ git config --global alias.unstage 'reset HEAD --'

It’s also common to add a last command, like this:

$ git config --global alias.last 'log -1 HEAD'

However, maybe you want to run an external command, rather than a Git subcommand. In that case, you start the command with a ! character. This is useful if you write your own tools that work with a Git repository. We can demonstrate by aliasing git visual to run gitk:

$ git config --global alias.visual '!gitk'

# 3 Git Branching

How does Git know what branch you’re currently on? It keeps a special pointer called HEAD. Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer to the local branch you’re currently on.

To switch to an existing branch, you run the git checkout command. Let’s switch to the new testing branch:

$ git checkout testing

This moves HEAD to point to the testing branch.

That command did two things. It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to. This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you’ve done in your testing branch so you can go in a different direction.

Because a branch in Git is actually a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

This resolution has a little of each section, and the <<<<<<<, =======, and >>>>>>> lines have been completely removed. After you’ve resolved each of these sections in each conflicted file, run git add on each file to mark it as resolved. Staging the file marks it as resolved in Git.

If you want to use a graphical tool to resolve these issues, you can run git mergetool, which fires up an appropriate visual merge tool and walks you through the conflicts.

You can run git status again to verify that all conflicts have been resolved:

**Branch Management**

To see the last commit on each branch, you can run git branch –v:

$ git branch –v

The useful --merged and --no-merged options can filter this list to branches that you have or have not yet merged into the branch you’re currently on. To see which branches are already merged into the branch you’re on, you can run git branch --merged:

$ git branch --merged

To see all the branches that contain work you haven’t yet merged in, you can run git branch --no-merged:

The options described above, --merged and --no-merged will, if not given a commit or branch name as an argument, show you what is, respectively, merged or not merged into your *current* branch.

You can always provide an additional argument to ask about the merge state with respect to some other branch without checking that other branch out first, as in, what is not merged into the master branch?

$ git checkout testing

$ git branch --no-merged master

**Remote Branches**

You can get a full list of remote references explicitly with git ls-remote [remote], or git remote show [remote] for remote branches as well as more information.

Remote-tracking branches are references to the state of remote branches. They’re local references that you can’t move; Git moves them for you whenever you do any network communication, to make sure they accurately represent the state of the remote repository. Think of them as bookmarks, to remind you where the branches in your remote repositories were the last time you connected to them.

Remote-tracking branches take the form <remote>/<branch>.

*“origin” is not special*

Just like the branch name “master” does not have any special meaning in Git, neither does “origin”. While “master” is the default name for a starting branch when you run git init which is the only reason it’s widely used, “origin” is the default name for a remote when you run git clone. If you run git clone -o booyah instead, then you will have booyah/master as your default remote branch.

To synchronize your work, you run a git fetch origin command. This command looks up which server “origin” is (in this case, it’s git.ourcompany.com), fetches any data from it that you don’t yet have, and updates your local database, moving your origin/master pointer to its new, more up-todate position.

**Pushing**

When you want to share a branch with the world, you need to push it up to a remote that you have write access to. Your local branches aren’t automatically synchronized to the remotes you write to — you have to explicitly push the branches you want to share. That way, you can use private branches for work you don’t want to share, and push up only the topic branches you want to collaborate on.

If you have a branch named serverfix that you want to work on with others, you can push it up the same way you pushed your first branch. Run git push <remote> <branch>:

$ git push origin serverfix

This is a bit of a shortcut. Git automatically expands the serverfix branchname out to refs/heads/serverfix:refs/heads/serverfix, which means, “Take my serverfix local branch and push it to update the remote’s serverfix branch.”

You can also do git push origin serverfix:serverfix, which does the same thing — it says, “Take my serverfix and make it the remote’s serverfix.” You can use this format to push a local branch into a remote branch that is named differently. If you didn’t want it to be called serverfix on the remote, you could instead run git push origin serverfix:awesomebranch to push your local serverfix branch to the awesomebranch branch on the remote project.

It’s important to note that when you do a fetch that brings down new remote-tracking branches, you don’t automatically have local, editable copies of them. In other words, in this case, you don’t have a new serverfix branch — you only have an origin/serverfix pointer that you can’t modify

To merge this work into your current working branch, you can run git merge origin/serverfix. If you want your own serverfix branch that you can work on, you can base it off your remotetracking branch:

$ git checkout -b serverfix origin/serverfix

This gives you a local branch that you can work on that starts where origin/serverfix is.

**Tracking Branches**

Checking out a local branch from a remote-tracking branch automatically creates what is called a “tracking branch” (and the branch it tracks is called an “upstream branch”). Tracking branches are local branches that have a direct relationship to a remote branch. If you’re on a tracking branch and type git pull, Git automatically knows which server to fetch from and which branch to merge in.

When you clone a repository, it generally automatically creates a master branch that tracks origin/master. However, you can set up other tracking branches if you wish — ones that track branches on other remotes, or don’t track the master branch. The simple case is the example you just saw, running git checkout -b <branch> <remote>/<branch>. This is a common enough operation that Git provides the --track shorthand:

$ git checkout --track origin/serverfix

In fact, this is so common that there’s even a shortcut for that shortcut. If the branch name you’re trying to checkout (a) doesn’t exist and (b) exactly matches a name on only one remote, Git will create a tracking branch for you:

$ git checkout serverfix

To set up a local branch with a different name than the remote branch, you can easily use the first version with a different local branch name:

$ git checkout -b sf origin/serverfix

Now, your local branch sf will automatically pull from origin/serverfix.

If you already have a local branch and want to set it to a remote branch you just pulled down, or want to change the upstream branch you’re tracking, you can use the -u or --set-upstream-to option to git branch to explicitly set it at any time.

$ git branch -u origin/serverfix

*Upstream shorthand*

When you have a tracking branch set up, you can reference its upstream branch with the @{upstream} or @{u} shorthand. So if you’re on the master branch and it’s tracking origin/master, you can say something like git merge @{u} instead of git merge origin/master if you wish.

If you want to see what tracking branches you have set up, you can use the -vv option to git branch. This will list out your local branches with more information including what each branch is tracking and if your local branch is ahead, behind or both.

**Pulling**

Generally it’s better to simply use the fetch and merge commands explicitly as the magic of git pull can often be confusing

**Deleting Remote Branches**

You can delete a remote branch using the --delete option to git push. If you want to delete your serverfix branch from the server, you run the following

$ git push origin --delete serverfix

**Rebasing**

In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase.

It works by going to the common ancestor of the two branches (the one you’re on and the one you’re rebasing onto), getting the diff introduced by each commit of the branch you’re on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

There is no difference in the end product of the integration, but rebasing makes for

a cleaner history. If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel

Note that the snapshot pointed to by the final commit you end up with, whether it’s the last of the rebased commits for a rebase or the final merge commit after a merge, is the same snapshot – it’s only the history that is different. Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together

**More Interesting Rebases**

****

Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes until it’s tested further. You can take the changes on client that aren’t on server (C8 and C9) and replay them on your master branch by using the --onto option of git rebase:

$ git rebase --onto master server client

This basically says, “Take the client branch, figure out the patches since it diverged from the server branch, and replay these patches in the client branch as if it was based directly off the master branch instead.” It’s a bit complex, but the result is pretty cool.

Let’s say you decide to pull in your server branch as well. You can rebase the server branch onto the master branch without having to check it out first by running git rebase <basebranch> <topicbranch> — which checks out the topic branch (in this case, server) for you and replays it onto the base branch (master):

$ git rebase master server

# 4 Git on the Server

**GitLab**

https://bitnami.com/stack/gitlab,

# 5 Distributed Git

The Git project has well-formatted commit messages — try running git log --no-merges there to see what a nicely-formatted project-commit history looks like.

The main difference is that merges happen client-side rather than on the server at commit time.

This is especially important to understand if you’re used to Subversion, because you’ll notice that the two developers didn’t edit the same file. Although Subversion automatically does such a merge on the server if different files are edited, with Git, you must *first* merge the commits locally. In other words, John must first fetch Jessica’s upstream changes and merge them into his local repository before he will

be allowed to push.

$ git log --no-merges issue54..origin/master

The issue54..origin/master syntax is a log filter that asks Git to display only those commits that are on the latter branch (in this case origin/master) that are not on the first branch (in this case issue54). We’ll go over this syntax in detail in Commit Ranges.

At this point, Jessica wants to push all of this merged “featureB” work back to the server, but she doesn’t want to simply push her own featureB branch. Rather, since Josie has already started an upstream featureBee branch, Jessica wants to push to *that* branch, which she does with:

$ git push -u origin featureB:featureBe

This is called a *refspec*. See The Refspec for a more detailed discussion of Git refspecs and different things you can do with them. Also notice the -u flag; this is short for --set-upstream, whichconfigures the branches for easier pushing and pulling later.

**Forked Public Project**

$ git clone <url>

$ cd project

$ git checkout -b featureA

... work ...

$ git commit

... work ...

$ git commit

……..

$ git remote add myfork <url>

$ git push -u myfork featureA

You start a new branch based off the current origin/master branch, squash the featureB changes there, resolve any conflicts, make the implementation change, and then push that as a new branch:

$ git checkout -b featureBv2 origin/master

$ git merge --squash featureB

... change implementation ...

$ git commit

$ git push myfork featureBv2



The --squash option takes all the work on the merged branch and squashes it into one changeset producing the repository state as if a real merge happened, without actually making a merge commit. This means your future commit will have one parent only and allows you to introduce all the changes from another branch and then make more changes before recording the new commit. Also the --no-commit option can be useful to delay the merge commit in case of the default merge process.

At this point, you can notify the maintainer that you’ve made the requested changes, and that they can find those changes in your featureBv2 branch.

**Determining What Is Introduced**

It’s often helpful to get a review of all the commits that are in this branch but that aren’t in your master branch. You can exclude commits in the master branch by adding the --not option before the branch name. This does the same thing as the master..contrib format that we used earlier. For example, if your contributor sends you two patches and you create a branch called contrib and applied those patches there, you can run this:

$ git log contrib --not master

To see what changes each commit introduces, remember that you can pass the -p option to git log and it will append the diff introduced to each commit.

To see a full diff of what would happen if you were to merge this topic branch with another branch, you may have to use a weird trick to get the correct results. You may think to run this:

$ git diff master

This command gives you a diff, but it may be misleading. If your master branch has moved forward since you created the topic branch from it, then you’ll get seemingly strange results. This happens because Git directly compares the snapshots of the last commit of the topic branch you’re on and the snapshot of the last commit on the master branch. For example, if you’ve added a line in a file on the master branch, a direct comparison of the snapshots will look like the topic branch is going to remove that line.

If master is a direct ancestor of your topic branch, this isn’t a problem; but if the two histories have diverged, the diff will look like you’re adding all the new stuff in your topic branch and removing everything unique to the master branch.

What you really want to see are the changes added to the topic branch — the work you’ll introduce if you merge this branch with master. You do that by having Git compare the last commit on your topic branch with the first common ancestor it has with the master branch.

Technically, you can do that by explicitly figuring out the common ancestor and then running your diff on it:

$ git merge-base contrib master

36c7dba2c95e6bbb78dfa822519ecfec6e1ca649

$ git diff 36c7db

or, more concisely:

$ git diff $(git merge-base contrib master)

However, neither of those is particularly convenient, so Git provides another shorthand for doing the same thing: the triple-dot syntax. In the context of the git diff command, you can put three periods after another branch to do a diff between the last commit of the branch you’re on and its common ancestor with another branch:

$ git diff master...contrib

This command shows you only the work your current topic branch has introduced since its common ancestor with master. That is a very useful syntax to remember.

**Rerere**

If you’re doing lots of merging and rebasing, or you’re maintaining a long-lived topic branch, Git has a feature called “rerere” that can help. Rerere stands for “reuse recorded resolution” — it’s a way of shortcutting manual conflict resolution. When rerere is enabled, Git will keep a set of pre- and post-images from successful merges, and if it notices that there’s a conflict that looks exactly like one you’ve already fixed, it’ll just use the fix from last time, without bothering you with it.

This feature comes in two parts: a configuration setting and a command. The configuration setting is rerere.enabled, and it’s handy enough to put in your global config:

$ git config --global rerere.enabled true

**Generating a Build Number**

Because Git doesn’t have monotonically increasing numbers like *v123* or the equivalent to go with each commit, if you want to have a human-readable name to go with a commit, you can run git describe on that commit. Git gives you the name of the nearest tag with the number of commits on top of that tag and a partial SHA-1 value of the commit you’re describing:

$ git describe master

This way, you can export a snapshot or build and name it something understandable to people. In fact, if you build Git from source code cloned from the Git repository, git --version gives you something that looks like this. If you’re describing a commit that you have directly tagged, it gives you the tag name.

The git describe command favors annotated tags (tags created with the -a or -s flag), so release tags should be created this way if you’re using git describe, to ensure the commit is named properly when described. You can also use this string as the target of a checkout or show command, although it relies on the abbreviated SHA-1 value at the end, so it may not be valid forever. For instance, the Linux kernel recently jumped from 8 to 10 characters to ensure SHA-1 object uniqueness, so older git describe output names were invalidated.

**Preparing a Release**

Now you want to release a build. One of the things you’ll want to do is create an archive of the latest snapshot of your code for those poor souls who don’t use Git. The command to do this is git archive:

$ git archive master --prefix='projet/' | gzip > `git describe master`.tar.gz

$ ls \*.tar.gz

v1.6.2-rc1-20-g8c5b85c.tar.gz

$ git archive master --prefix='project/' --format=zip > `git describe master`.zip

**Shortlog**

It’s time to email your mailing list of people who want to know what’s happening in your project. A nice way of quickly getting a sort of changelog of what has been added to your project since your last release or email is to use the git shortlog command. It summarizes all the commits in the range you give it; for example, the following gives you a summary of all the commits since your last release, if your last release was named v1.0.1:

$ git shortlog --no-merges master --not v1.0.1

# 6 GitHub

To demonstrate this, we’re going to use a low-level command (often referred to as a “plumbing”command, which we’ll read about more in Plumbing and Porcelain) called ls-remote. This command is generally not used in day-to-day Git operations but it’s useful to show us what references are present on the server.

If we run this command against the “blink” repository we were using earlier, we will get a list of all the branches and tags and other references in the repository.

# 7 Git Tools

Git can figure out a short, unique abbreviation for your SHA-1 values. If you pass --abbrev-commit to the git log command, the output will use shorter values but keep them unique; it defaults to using seven characters but makes them longer if necessary to keep the SHA-1 unambiguous:

$ git log –abrev-commit –pretty=oneline

If you want to see which specific SHA-1 a branch points to, or if you want to see what any of these examples boils down to in terms of SHA-1s, you can use a Git plumbing tool called rev-parse

$ git rev-parse topic1

One of the things Git does in the background while you’re working away is keep a “reflog” — a log of where your HEAD and branch references have been for the last few months. You can see your reflog by using git reflog:

If you want to see the fifth prior value of the HEAD of your repository, you can use the @{5} reference that you see in the reflog output:

$ git show HEAD@{5}

To see reflog information formatted like the git log output, you can run git log -g

It’s important to note that reflog information is strictly local — it’s a log only of what *you’ve* done in *your* repository.

**Ancestry References**

The other main way to specify a commit is via its ancestry. If you place a ^ (caret) at the end of a reference, Git resolves it to mean the parent of that commit

Then, you can see the previous commit by specifying HEAD^, which means “the parent of HEAD”:

You can also specify a number after the ^ – for example, d921970^2 means “the second parent of d921970.” This syntax is useful only for merge commits, which have more than one parent. The first parent is the branch you were on when you merged, and the second is the commit on the branch that you merged in:

The other main ancestry specification is the ~ (tilde). This also refers to the first parent, so HEAD~ and HEAD^ are equivalent. The difference becomes apparent when you specify a number. HEAD~2 means “the first parent of the first parent,” or “the grandparent” — it traverses the first parents the number of times you specify. For example, in the history listed earlier.

HEAD~3 peut aussi s’ecrire HEAD^^^.

You can also combine these syntaxes — you can get the second parent of the previous reference (assuming it was a merge commit) by using HEAD~3^2, and so on

**Commit Ranges**

**Double Dot**

Say you want to see what is in your experiment branch that hasn’t yet been merged into your master branch. You can ask Git to show you a log of just those commits with master..experiment — that means “all commits reachable from experiment that aren’t reachable from master.” For the sake of brevity and clarity in these examples, the letters of the commit objects from the diagram are used in place of the actual log output in the order that they would display:

$ git log master..experiment

$ git log origin/master..HEAD

This command shows you any commits in your current branch that aren’t in the master branch on your origin remote. If you run a git push and your current branch is tracking origin/master, the commits listed by git log origin/master..HEAD are the commits that will be transferred to the server. You can also leave off one side of the syntax to have Git assume HEAD. For example, you can get the same results as in the previous example by typing git log origin/master.. — Git substitutes HEAD if one side is missing.

**Multiple Points**

The double-dot syntax is useful as a shorthand, but perhaps you want to specify more than two branches to indicate your revision, such as seeing what commits are in any of several branches that aren’t in the branch you’re currently on. Git allows you to do this by using either the ^ character or --not before any reference from which you don’t want to see reachable commits. Thus, the following three commands are equivalent:

$ git log refA..refB

$ git log ^refA refB

$ git log refB --not refA

This is nice because with this syntax you can specify more than two references in your query, which you cannot do with the double-dot syntax. For instance, if you want to see all commits that are reachable from refA or refB but not from refC, you can use either of:

$ git log refA refB ^refC

$ git log refA refB --not refC

**Triple Dot**

The last major range-selection syntax is the triple-dot syntax, which specifies all the commits that are reachable by *either* of two references but not by both of them

$ git log master...experiment

A common switch to use with the log command in this case is --left-right, which shows you which side of the range each commit is in. This helps make the output more useful

$ git log --left-right master...experiment

With these tools, you can much more easily let Git know what commit or commits you want to inspect.

**Interactive Staging**

If you run git add with the -i or --interactive option, Git enters an interactive shell mode

Patch Mode

$ git add –p

$ git reset –p

$ git checkout –p

$ git stash save -p

**Stashing and Cleaning**

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time (even on a different branch).

$ git stash save

$ git stash list

*Migrating to* git stash push

As of late October 2017, there has been extensive discussion on the Git mailing list,

wherein the command git stash save is being deprecated in favour of the existing

alternative git stash push. The main reason for this is that git stash push

introduces the option of stashing selected *pathspecs*, something git stash save

does not support

To push a new stash onto your stack, run git stash or git stash save:

To see which stashes you’ve stored, you can use git stash list

You can reapply the one you just stashed by using the command shown in the help output of the original stash command: git stash apply. If you want to apply one of the older stashes, you can specify it by naming it, like this: git stash apply stash@{2}. If you don’t specify a stash, Git assumes the most recent stash and tries to apply it:

$ git stash apply

The changes to your files were reapplied, but the file you staged before wasn’t restaged. To do that, you must run the git stash apply command with a --index option to tell the command to try to reapply the staged changes.

To remove it, you can run git stash drop with the name of the stash to remove. You can also run git stash pop to apply the stash and then immediately drop it from your stack.

**Creative Stashing**

There are a few stash variants that may also be helpful. The first option that is quite popular is the --keep-index option to the stash save command. This tells Git to not only include all staged content in the stash being created, but simultaneously leave it in the index.

Another common thing you may want to do with stash is to stash the untracked files as well as the tracked ones. By default, git stash will stash only modified and staged *tracked* files. If you specify --include-untracked or -u, Git will include untracked files in the stash being created.

Finally, if you specify the --patch flag, Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory.

**Creating a Branch from a Stash**

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work. If the apply tries to modify a file that you’ve since modified, you’ll get a merge conflict and will have to try to resolve it. If you want an easier way to test the stashed changes again, you can run git stash branch <branch>, which creates a new branch for you with your selected branch name, checks out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully:

$ git stash branch testchanges

**Reset Demystified**

These commands are two of the most confusing parts of Git when you first encounter them. They do so many things that it seems hopeless to actually understand them and employ them properly.

Git as a system manages and manipulates three trees in its normal operation:

**Tree Role**

HEAD Last commit snapshot, next parent

Index Proposed next commit snapshot

Working Directory Sandbox

**The HEAD**

HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch. That means HEAD will be the parent of the next commit that is created. It’s generally simplest to think of HEAD as the snapshot of **your last commit on that branch**..

$ git cat-file -p HEAD

$ git ls-tree -r HEAD

The Git cat-file and ls-tree commands are “plumbing” commands that are used for lower level things and not really used in day-to-day work, but they help us see what’s going on here.

**The Index**

The Index is your **proposed next commit**. We’ve also been referring to this concept as Git’s “Staging Area” as this is what Git looks at when you run git commit.

Git populates this index with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out. You then replace some of those files with new versions of them, and git commit converts that into the tree for a new commit.

$ git ls-files –s

Again, here we’re using git ls-files, which is more of a behind the scenes command that shows you what your index currently looks like.

**The Working Directory**

Finally, you have your working directory. The other two trees store their content in an efficient but

inconvenient manner, inside the .git folder. The Working Directory unpacks them into actual files, which makes it much easier for you to edit them. Think of the Working Directory as a **sandbox**, where you can try changes out before committing them to your staging area (index) and then to history.

**The Workflow**

Git’s main purpose is to record snapshots of your project in successively better states, by manipulating these three trees.



Switching branches or cloning goes through a similar process. When you checkout a branch, it changes **HEAD** to point to the new branch ref, populates your **Index** with the snapshot of that commit, then copies the contents of the **Index** into your **Working Directory**.

**The Role of Reset**

The reset command makes more sense when viewed in this context.

Let’s now walk through exactly what reset does when you call it. It directly manipulates these three trees in a simple and predictable way. It does up to three basic operations.

**Step 1: Move HEAD**

The first thing reset will do is move what HEAD points to. This isn’t the same as changing HEAD itself (which is what checkout does); reset moves the branch that HEAD is pointing to.

No matter what form of reset with a commit you invoke, this is the first thing it will always try to do. With reset --soft, it will simply stop there.

**Step 2: Updating the Index (--mixed))**

The next thing reset will do is to update the Index with the contents of whatever snapshot HEAD now points to.

If you specify the --mixed option, reset will stop at this point. This is also the default, so if you specify no option at all (just git reset HEAD~ in this case), this is where the command will stop.

**Step 3: Updating the Working Directory (--hard)**

The third thing that reset will do is to make the Working Directory look like the Index. If you use the --hard option, it will continue to this stage.

It’s important to note that this flag (--hard) is the only way to make the reset command dangerous, and one of the very few cases where Git will actually destroy data. Any other invocation of reset can be pretty easily undone, but the --hard option cannot

**Recap**

The reset command overwrites these three trees in a specific order, stopping when you tell it to:

1. Move the branch HEAD points to *(stop here if* --soft*)*

2. Make the Index look like HEAD *(stop here unless* --hard*)*

3. Make the Working Directory look like the Index

**Reset With a Path**

So, assume we run git reset file.txt. This form (since you did not specify a commit SHA-1 or branch, and you didn’t specify --soft or --hard) is shorthand for git reset --mixed HEAD file.txt,

So it essentially just copies file.txt from HEAD to the Index.

This has the practical effect of *unstaging* the file.

This is why the output of the git status command suggests that you run this to unstage a file. (SeeUnstaging a Staged File for more on this.)

We could just as easily not let Git assume we meant “pull the data from HEAD” by specifying a specific commit to pull that file version from. We would just run something like git reset eb43bf file.txt.

**Squashing**

You can run git reset --soft HEAD~2 to move the HEAD branch back to an older commit (the mostrecent commit you want to keep):

**Check It Out**

Finally, you may wonder what the difference between checkout and reset is. Like reset, checkout manipulates the three trees, and it is a bit different depending on whether you give the command a file path or not.

**Without Paths**

Running git checkout [branch] is pretty similar to running git reset --hard [branch] in that it updates all three trees for you to look like [branch], but there are two important differences.

First, unlike reset --hard, checkout is working-directory safe; it will check to make sure it’s not blowing away files that have changes to them. Actually, it’s a bit smarter than that — it tries to do a trivial merge in the Working Directory, so all of the files you *haven’t* changed will be updated. Reset --hard, on the other hand, will simply replace everything across the board without checking.

The second important difference is how checkout updates HEAD. Whereas reset will move the branch that HEAD points to, checkout will move HEAD itself to point to another branch.



**With Paths**

The other way to run checkout is with a file path, which, like reset, does not move HEAD. It is just like git reset [branch] file in that it updates the index with that file at that commit, but it also overwrites the file in the working directory. It would be exactly like git reset --hard [branch] file (if reset would let you run that) — it’s not working-directory safe, and it does not move HEAD.

Also, like git reset and git add, checkout will accept a --patch option to allow you to selectively revert file contents on a hunk-by-hunk basis.

**Summary**

Here’s a cheat-sheet for which commands affect which trees. The “HEAD” column reads “REF” if that command moves the reference (branch) that HEAD points to, and “HEAD” if it moves HEAD itself. Pay especial attention to the *WD Safe?* column — if it says **NO**, take a second to think before running that command.



**Advanced Merging**

**Merge Conflicts**

**Ignoring Whitespace**

The default merge strategy can take arguments though, and a few of them are about properly ignoring whitespace changes. If you see that you have a lot of whitespace issues in a merge, you can simply abort it and do it again, this time with -Xignore-all-space or -Xignore-space-change. The first option ignores whitespace **completely** when comparing lines, the second treats sequences of one or more whitespace characters as equivalent

$ git merge -Xignore-space-change branch1

First, we get into the merge conflict state. Then we want to get copies of my version of the file, their version (from the branch we’re merging in) and the common version (from where both sides branched off). Then we want to fix up either their side or our side and re-try the merge again for just this single file.

Getting the three file versions is actually pretty easy. Git stores all of these versions in the index under “stages” which each have numbers associated with them. Stage 1 is the common ancestor, stage 2 is your version and stage 3 is from the MERGE\_HEAD, the version you’re merging in (“theirs”).

You can extract a copy of each of these versions of the conflicted file with the git show command and a special syntax.

$ git show :1:hello.rb > hello.common.rb

$ git show :2:hello.rb > hello.ours.rb

$ git show :3:hello.rb > hello.theirs.rb

If you want to get a little more hard core, you can also use the ls-files -u plumbing command to get the actual SHA-1s of the Git blobs for each of these files.

$ git ls-files -u

The :1:hello.rb is just a shorthand for looking up that blob SHA-1.

Now that we have the content of all three stages in our working directory, we can manually fix up theirs to fix the whitespace issue and re-merge the file with the little-known git merge-file command which does just that.

$ git merge-file -p hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

If you want to get an idea before finalizing this commit about what was actually changed between one side or the other, you can ask git diff to compare what is in your working directory that you’re about to commit as the result of the merge to any of these stages. Let’s go through them all.

To compare your result to what you had in your branch before the merge, in other words, to see what the merge introduced, you can run git diff –ours

If we want to see how the result of the merge differed from what was on their side, you can run git diff --theirs.

Finally, you can see how the file has changed from both sides with git diff --base.

**Checking Out Conflicts**

If we open up the file, we’ll see something like this:

#! /usr/bin/env ruby

def hello

<<<<<<< HEAD

puts 'hola world'

======

puts 'hello mundo'

>>>>>>> mundo

end

hello()

Perhaps it’s not obvious how exactly you should fix this conflict. You need more

context.

One helpful tool is git checkout with the ‘--conflict’ option. This will re-checkout the file again and replace the merge conflict markers. This can be useful if you want to reset the markers and try to resolve them again.

You can pass --conflict either diff3 or merge (which is the default). If you pass it diff3, Git will use a slightly different version of conflict markers, not only giving you the “ours” and “theirs” versions, but also the “base” version inline to give you more context.

$ git checkout --conflict=diff3 hello.rb

Une fois que nous l’avons lance, le fichier ressemble a ceci :

#! /usr/bin/env ruby

def hello

<<<<<<< ours

puts 'hola world'

||||||| base

puts 'hello world'

======

puts 'hello mundo'

>>>>>>> theirs

end

hello()

If you like this format, you can set it as the default for future merge conflicts by setting the merge.conflictstyle setting to diff3..

$ git config --global merge.conflictstyle diff3

The git checkout command can also take --ours and --theirs options, which can be a really fast way of just choosing either one side or the other without merging things at all.

This can be particularly useful for conflicts of binary files where you can simply choose one side, or where you only want to merge certain files in from another branch - you can do the merge and then checkout certain files from one side or the other before committing

**Merge Log**

Another useful tool when resolving merge conflicts is git log. This can help you get context on what may have contributed to the conflicts. Reviewing a little bit of history to remember why two lines of development were touching the same area of code can be really helpful sometimes.

To get a full list of all of the unique commits that were included in either branch involved in this merge, we can use the “triple dot” syntax that we learned in Triple Dot.

$ git log --oneline --left-right HEAD...MERGE\_HEAD

We can further simplify this though to give us much more specific context. If we add the --merge option to git log, it will only show the commits in either side of the merge that touch a file that’s currently conflicted.

$ git log --oneline --left-right –merge

If you run that with the -p option instead, you get just the diffs to the file that ended up in conflict.This can be **really** helpful in quickly giving you the context you need to help understand why something conflicts and how to more intelligently resolve it.

**Combined Diff Format**

When you run git diff directly after a merge conflict, it will give you information in a rather unique diff output format.

$ git diff

diff --cc hello.rb

index 0399cd5,59727f0..0000000

--- a/hello.rb

+++ b/hello.rb

@@@ -1,7 -1,7 +1,11 @@@

#! /usr/bin/env ruby

def hello

++<<<<<<< HEAD

+ puts 'hola world'

++=======

+ puts 'hello mundo'

++>>>>>>> mundo

end

hello()

The format is called “Combined Diff” and gives you two columns of data next to each line. The first column shows you if that line is different (added or removed) between the “ours” branch and the file in your working directory and the second column does the same between the “theirs” branch and your working directory copy.

So in that example you can see that the <<<<<<< and >>>>>>> lines are in the working copy but were not in either side of the merge. This makes sense because the merge tool stuck them in there for our context, but we’re expected to remove them.

You can also get this from the git log for any merge to see how something was resolved after the fact. Git will output this format if you run git show on a merge commit, or if you add a --cc option to a git log -p (which by default only shows patches for non-merge commits).

$ git log --cc -p -1

**Undoing Merges**

**Fix the references**

$ git reset --hard HEAD~

The downside of this approach is that it’s rewriting history, which can be problematic with a shared repository. Check out The Perils of Rebasing for more on what can happen; the short version is that if other people have the commits you’re rewriting, you should probably avoid reset. This approach also won’t work if any other commits have been created since the merge; moving the refs would effectively lose those changes.

**Reverse the commit**

If moving the branch pointers around isn’t going to work for you, Git gives you the option of making a new commit which undoes all the changes from an existing one. Git calls this operation a “revert”, and in this particular scenario, you’d invoke it like this:

$ git revert -m 1 HEAD

The -m 1 flag indicates which parent is the “mainline” and should be kept.

**Other Types of Merges**

**Our or Theirs Preference**

First of all, there is another useful thing we can do with the normal “recursive” mode of merging. We’ve already seen the ignore-all-space and ignore-space-change options which are passed with a -X but we can also tell Git to favor one side or the other when it sees a conflict.

By default, when Git sees a conflict between two branches being merged, it will add merge conflict markers into your code and mark the file as conflicted and let you resolve it. If you would prefer for Git to simply choose a specific side and ignore the other side instead of letting you manually resolve the conflict, you can pass the merge command either a -Xours or -Xtheirs

Si vous preferez que Git choisisse simplement un cote specifique et qu’il ignore l’autre cote au lieu de vous laisser fusionner manuellement le conflit, vous pouvez passer -Xours ou -Xtheirs a la commande merge.

If Git sees this, it will not add conflict markers. Any differences that are mergeable, it will merge. Any differences that conflict, it will simply choose the side you specify in whole, including binary files.

$ git merge -Xours mundo

This option can also be passed to the git merge-file command we saw earlier by running something like git merge-file --ours for individual file merges

If you want to do something like this but not have Git even try to merge changes from the other side in, there is a more draconian option, which is the “ours” merge *strategy*. This is different from the “ours” recursive merge *option*.

This will basically do a fake merge. It will record a new merge commit with both branches as parents, but it will not even look at the branch you’re merging in. It will simply record as the result of the merge the exact code in your current branch.

$ git merge -s ours mundo

$ git diff HEAD HEAD~

$

You can see that there is no difference between the branch we were on and the result of the merge

**Rerere**

The git rerere functionality is a bit of a hidden feature. The name stands for “reuse recorded resolution” and, as the name implies, it allows you to ask Git to remember how you’ve resolved a hunk conflict so that the next time it sees the same conflict, Git can resolve it for you automatically

**Debugging with Git**

**Maintenance and Data Recovery**

Occasionally, you may have to do some cleanup – make a repository more compact, clean up an imported repository, or recover lost work. This section will cover some of these scenarios.

**Maintenance**

Occasionally, Git automatically runs a command called “auto gc”. Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged git gc command. The “gc” stands for garbage collect, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren’t reachable from any commit and are a few months old.

$ git gc

**Data Recovery**

Often, the quickest way is to use a tool called git reflog. As you’re working, Git silently records what your HEAD is every time you change it. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the git update-ref command, which is another reason to use it instead of just writing the SHA-1 value to your ref files, as we covered in Git References. You can see where you’ve been at any time by running git reflog:

$ git reflog

1a410ef HEAD@{0}: reset: moving to 1a410ef

ab1afef HEAD@{1}: commit: modified repo.rb a bit

484a592 HEAD@{2}: commit: added repo.rb

To see the same information in a much more useful way, we can run git log -g, which will give you a normal log output for your reflog

Because the reflog data is kept in the .git/logs/ directory, you effectively have no reflog. How can you recover that commit at this point? One way is to use the git fsck utility, which checks your database for integrity. If you run it with the --full option, it shows you all objects that aren’t pointed to by another object:

$ git fsck --full

Checking object directories: 100% (256/256), done.

Checking objects: 100% (18/18), done.

dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4

dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b

dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9

dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293

Vous pouvez executer la commande count-objects pour voir rapidement combien d’espace disque vous utilisez :

$ git count-objects -v

# Appendix A: Git in Other Environments

git-gui, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

$ git gui

gitk and git-gui are examples of task-oriented tools. Each of them is tailored for a specific purpose (viewing history and creating commits, respectively), and omit the features not necessary for that task.