### Git Basics

* Introduction to Git

So, what is Git in a nutshell? This is an important section to absorb, because if you understand what Git is and the fundamentals of how it works, then using Git effectively will probably be much easier for you. As you learn Git, try to clear your mind of the things you may know about other VCSs, such as CVS, Subversion or Perforce — doing so will help you avoid subtle confusion when using the tool. Even though Git’s user interface is fairly similar to these other VCSs, Git stores and thinks about information in a very different way, and understanding these differences will help you avoid becoming confused while using it.

### **Snapshots, Not Differences**

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These other systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they store as a set of files and the changes made to each file over time (this is commonly described as **delta-based** version control).



Figure 4. Storing data as changes to a base version of each file

Git doesn’t think of or store its data this way. Instead, Git thinks of its data more like a series of snapshots of a miniature filesystem. With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**.



Figure 5. Storing data as snapshots of the project over time

This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation. This makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS.

### **Nearly Every Operation Is Local**

Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on your network. If you’re used to a CVCS where most operations have that network latency overhead, this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

For example, to browse the history of the project, Git doesn’t need to go out to the server to get the history and display it for you — it simply reads it directly from your local database. This means you see the project history almost instantly. If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.

This also means that there is very little you can’t do if you’re offline or off VPN. If you get on an airplane or a train and want to do a little work, you can commit happily (to your **local** copy, remember?) until you get to a network connection to upload. If you go home and can’t get your VPN client working properly, you can still work. In many other systems, doing so is either impossible or painful. In Perforce, for example, you can’t do much when you aren’t connected to the server; in Subversion and CVS, you can edit files, but you can’t commit changes to your database (because your database is offline). This may not seem like a huge deal, but you may be surprised what a big difference it can make.

### **Git Has Integrity**

Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it’s impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can’t lose information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

24b9da6552252987aa493b52f8696cd6d3b00373

You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything in its database not by file name but by the hash value of its contents.

### **Git Generally Only Adds Data**

When you do actions in Git, nearly all of them only **add** data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. As with any VCS, you can lose or mess up changes you haven’t committed yet, but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

This makes using Git a joy because we know we can experiment without the danger of severely screwing things up.

### 

### **The Three States**

Pay attention now — here is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: **modified**, **staged**, and **committed**:

* Modified means that you have changed the file but have not committed it to your database yet.
* Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
* Committed means that the data is safely stored in your local database.

This leads us to the three main sections of a Git project: the working tree, the staging area, and the Git directory.



Figure 6. Working tree, staging area, and Git directory

The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you **clone** a repository from another computer.

The basic Git workflow goes something like this:

1. You modify files in your working tree.
2. You selectively stage just those changes you want to be part of your next commit, which adds **only** those changes to the staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

If a particular version of a file is in the Git directory, it’s considered **committed**. If it has been modified and was added to the staging area, it is **staged**. And if it was changed since it was checked out but has not been staged, it is **modified**.

* Git Repository

You typically obtain a Git repository in one of two ways:

1. You can take a local directory that is currently not under version control, and turn it into a Git repository, or
2. You can **clone** an existing Git repository from elsewhere.

In either case, you end up with a Git repository on your local machine, ready for work.

### **Initializing a Repository in an Existing Directory**

If you have a project directory that is currently not under version control and you want to start controlling it with Git, you first need to go to that project’s directory. If you’ve never done this, it looks a little different depending on which system you’re running:

for Linux:

$ cd /home/user/my\_project

for macOS:

$ cd /Users/user/my\_project

for Windows:

$ cd C:/Users/user/my\_project

and type:

$ git init

This creates a new subdirectory named .git that contains all of your necessary repository files — a Git repository skeleton. At this point, nothing in your project is tracked yet. See [Git Internals](https://git-scm.com/book/en/v2/ch00/ch10-git-internals) for more information about exactly what files are contained in the .git directory you just created.

If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few git add commands that specify the files you want to track, followed by a git commit:

$ git add \*.c

$ git add LICENSE

$ git commit -m 'Initial project version'

We’ll go over what these commands do in just a minute. At this point, you have a Git repository with tracked files and an initial commit.

### **Cloning an Existing Repository**

If you want to get a copy of an existing Git repository — for example, a project you’d like to contribute to — the command you need is git clone. If you’re familiar with other VCSs such as Subversion, you’ll notice that the command is "clone" and not "checkout". This is an important distinction — instead of getting just a working copy, Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run git clone. In fact, if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned (you may lose some server-side hooks and such, but all the versioned data would be there.

You clone a repository with git clone <url>. For example, if you want to clone the Git linkable library called libgit2, you can do so like this:

$ git clone https://github.com/libgit2/libgit2

That creates a directory named libgit2, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new libgit2 directory that was just created, you’ll see the project files in there, ready to be worked on or used.

If you want to clone the repository into a directory named something other than libgit2, you can specify the new directory name as an additional argument:

$ git clone https://github.com/libgit2/libgit2 mylibgit

That command does the same thing as the previous one, but the target directory is called mylibgit.

Git has a number of different transfer protocols you can use. The previous example uses the https:// protocol, but you may also see git:// or user@server:path/to/repo.git, which uses the SSH transfer protocol.

* Recording Changes to the Repository

At this point, you should have a **bona fide** Git repository on your local machine, and a checkout or **working copy** of all of its files in front of you. Typically, you’ll want to start making changes and committing snapshots of those changes into your repository each time the project reaches a state you want to record.

Remember that each file in your working directory can be in one of two states: **tracked** or **untracked**. Tracked files are files that were in the last snapshot, as well as any newly staged files; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.

Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.

As you edit files, Git sees them as modified, because you’ve changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.



Figure 8. The lifecycle of the status of your files

### **Checking the Status of Your Files**

The main tool you use to determine which files are in which state is the git status command. If you run this command directly after a clone, you should see something like this:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean

This means you have a clean working directory; in other words, none of your tracked files are modified. Git also doesn’t see any untracked files, or they would be listed here. Finally, the command tells you which branch you’re on and informs you that it has not diverged from the same branch on the server. For now, that branch is always master, which is the default; you won’t worry about it here. [Git Branching](https://git-scm.com/book/en/v2/ch00/ch03-git-branching) will go over branches and references in detail.

Let’s say you add a new file to your project, a simple README file. If the file didn’t exist before, and you run git status, you see your untracked file like so:

$ echo 'My Project' > README

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

README

nothing added to commit but untracked files present (use "git add" to track)

You can see that your new README file is untracked, because it’s under the “Untracked files” heading in your status output. Untracked basically means that Git sees a file you didn’t have in the previous snapshot (commit), and which hasn’t yet been staged; Git won’t start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don’t accidentally begin including generated binary files or other files that you did not mean to include. You do want to start including README, so let’s start tracking the file.

### **Tracking New Files**

In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:

$ git add README

If you run your status command again, you can see that your README file is now tracked and staged to be committed:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: README

You can tell that it’s staged because it’s under the “Changes to be committed” heading. If you commit at this point, the version of the file at the time you ran git add is what will be in the subsequent historical snapshot. You may recall that when you ran git init earlier, you then ran git add <files> — that was to begin tracking files in your directory. The git add command takes a path name for either a file or a directory; if it’s a directory, the command adds all the files in that directory recursively.

### **Staging Modified Files**

Let’s change a file that was already tracked. If you change a previously tracked file called CONTRIBUTING.md and then run your git status command again, you get something that looks like this:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

The CONTRIBUTING.md file appears under a section named “Changes not staged for commit” — which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the git add command. git add is a multipurpose command — you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add precisely this content to the next commit” rather than “add this file to the project”. Let’s run git add now to stage the CONTRIBUTING.md file, and then run git status again:

$ git add CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

Both files are staged and will go into your next commit. At this point, suppose you remember one little change that you want to make in CONTRIBUTING.md before you commit it. You open it again and make that change, and you’re ready to commit. However, let’s run git status one more time:

$ vim CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

What the heck? Now CONTRIBUTING.md is listed as both staged **and** unstaged. How is that possible? It turns out that Git stages a file exactly as it is when you run the git add command. If you commit now, the version of CONTRIBUTING.md as it was when you last ran the git add command is how it will go into the commit, not the version of the file as it looks in your working directory when you run git commit. If you modify a file after you run git add, you have to run git add again to stage the latest version of the file:

$ git add CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

### **Short Status**

While the git status output is pretty comprehensive, it’s also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simplified output from the command:

$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

New files that aren’t tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on. There are two columns to the output — the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

### **Ignoring Files**

Often, you’ll have a class of files that you don’t want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named .gitignore. Here is an example .gitignore file:

$ cat .gitignore

\*.[oa]

\*~

The first line tells Git to ignore any files ending in “.o” or “.a” — object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a .gitignore file for your new repository before you get going is generally a good idea so you don’t accidentally commit files that you really don’t want in your Git repository.

The rules for the patterns you can put in the .gitignore file are as follows:

* Blank lines or lines starting with # are ignored.
* Standard glob patterns work, and will be applied recursively throughout the entire working tree.
* You can start patterns with a forward slash (/) to avoid recursivity.
* You can end patterns with a forward slash (/) to specify a directory.
* You can negate a pattern by starting it with an exclamation point (!).

Glob patterns are like simplified regular expressions that shells use. An asterisk (\*) matches zero or more characters; [abc] matches any character inside the brackets (in this case a, b, or c); a question mark (?) matches a single character; and brackets enclosing characters separated by a hyphen ([0-9]) matches any character between them (in this case 0 through 9). You can also use two asterisks to match nested directories; a/\*\*/z would match a/z, a/b/z, a/b/c/z, and so on.

Here is another example .gitignore file:

# ignore all .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in any directory named build

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories

doc/\*\*/\*.pdf

| **Tip** | GitHub maintains a fairly comprehensive list of good .gitignore file examples for dozens of projects and languages at <https://github.com/github/gitignore> if you want a starting point for your project. |
| --- | --- |

| **Note** | In the simple case, a repository might have a single .gitignore file in its root directory, which applies recursively to the entire repository. However, it is also possible to have additional .gitignore files in subdirectories. The rules in these nested .gitignore files apply only to the files under the directory where they are located. The Linux kernel source repository has 206 .gitignore files.  It is beyond the scope of this book to get into the details of multiple .gitignore files; see man gitignore for the details. |
| --- | --- |

### **Viewing Your Staged and Unstaged Changes**

If the git status command is too vague for you — you want to know exactly what you changed, not just which files were changed — you can use the git diff command. We’ll cover git diff in more detail later, but you’ll probably use it most often to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit? Although git status answers those questions very generally by listing the file names, git diff shows you the exact lines added and removed — the patch, as it were.

Let’s say you edit and stage the README file again and then edit the CONTRIBUTING.md file without staging it. If you run your git status command, you once again see something like this:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

To see what you’ve changed but not yet staged, type git diff with no other arguments:

$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

That command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

If you want to see what you’ve staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit:

$ git diff --staged

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

+My Project

It’s important to note that git diff by itself doesn’t show all changes made since your last commit — only changes that are still unstaged. If you’ve staged all of your changes, git diff will give you no output.

For another example, if you stage the CONTRIBUTING.md file and then edit it, you can use git diff to see the changes in the file that are staged and the changes that are unstaged. If our environment looks like this:

$ git add CONTRIBUTING.md

$ echo '# test line' >> CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: CONTRIBUTING.md

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

Now you can use git diff to see what is still unstaged:

$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 643e24f..87f08c8 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -119,3 +119,4 @@ at the

## Starter Projects

See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).

+# test line

and git diff --cached to see what you’ve staged so far (--staged and --cached are synonyms):

$ git diff --cached

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

| **Note** | **Git Diff in an External Tool**  We will continue to use the git diff command in various ways throughout the rest of the book. There is another way to look at these diffs if you prefer a graphical or external diff viewing program instead. If you run git difftool instead of git diff, you can view any of these diffs in software like emerge, vimdiff and many more (including commercial products). Run git difftool --tool-help to see what is available on your system. |
| --- | --- |

### **Committing Your Changes**

Now that your staging area is set up the way you want it, you can commit your changes. Remember that anything that is still unstaged — any files you have created or modified that you haven’t run git add on since you edited them — won’t go into this commit. They will stay as modified files on your disk. In this case, let’s say that the last time you ran git status, you saw that everything was staged, so you’re ready to commit your changes. The simplest way to commit is to type git commit:

$ git commit

Doing so launches your editor of choice.

| **Note** | This is set by your shell’s EDITOR environment variable — usually vim or emacs, although you can configure it with whatever you want using the git config --global core.editor command as you saw in [Getting Started](https://git-scm.com/book/en/v2/ch00/ch01-getting-started). |
| --- | --- |

The editor displays the following text (this example is a Vim screen):

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Your branch is up-to-date with 'origin/master'.

#

# Changes to be committed:

# new file: README

# modified: CONTRIBUTING.md

#

~

~

~

".git/COMMIT\_EDITMSG" 9L, 283C

You can see that the default commit message contains the latest output of the git status command commented out and one empty line on top. You can remove these comments and type your commit message, or you can leave them there to help you remember what you’re committing.

| **Note** | For an even more explicit reminder of what you’ve modified, you can pass the -v option to git commit. Doing so also puts the diff of your change in the editor so you can see exactly what changes you’re committing. |
| --- | --- |

When you exit the editor, Git creates your commit with that commit message (with the comments and diff stripped out).

Alternatively, you can type your commit message inline with the commit command by specifying it after a -m flag, like this:

$ git commit -m "Story 182: fix benchmarks for speed"

[master 463dc4f] Story 182: fix benchmarks for speed

2 files changed, 2 insertions(+)

create mode 100644 README

Now you’ve created your first commit! You can see that the commit has given you some output about itself: which branch you committed to (master), what SHA-1 checksum the commit has (463dc4f), how many files were changed, and statistics about lines added and removed in the commit.

Remember that the commit records the snapshot you set up in your staging area. Anything you didn’t stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you’re recording a snapshot of your project that you can revert to or compare to later.

### **Skipping the Staging Area**

Although it can be amazingly useful for crafting commits exactly how you want them, the staging area is sometimes a bit more complex than you need in your workflow. If you want to skip the staging area, Git provides a simple shortcut. Adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")

$ git commit -a -m 'Add new benchmarks'

[master 83e38c7] Add new benchmarks

1 file changed, 5 insertions(+), 0 deletions(-)

Notice how you don’t have to run git add on the CONTRIBUTING.md file in this case before you commit. That’s because the -a flag includes all changed files. This is convenient, but be careful; sometimes this flag will cause you to include unwanted changes.

### **Removing Files**

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The git rm command does that, and also removes the file from your working directory so you don’t see it as an untracked file the next time around.

If you simply remove the file from your working directory, it shows up under the “Changes not staged for commit” (that is, **unstaged**) area of your git status output:

$ rm PROJECTS.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")

Then, if you run git rm, it stages the file’s removal:

$ git rm PROJECTS.md

rm 'PROJECTS.md'

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

deleted: PROJECTS.md

The next time you commit, the file will be gone and no longer tracked. If you modified the file or had already added it to the staging area, you must force the removal with the -f option. This is a safety feature to prevent accidental removal of data that hasn’t yet been recorded in a snapshot and that can’t be recovered from Git.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your .gitignore file and accidentally staged it, like a large log file or a bunch of .a compiled files. To do this, use the --cached option:

$ git rm --cached README

You can pass files, directories, and file-glob patterns to the git rm command. That means you can do things such as:

$ git rm log/\\*.log

Note the backslash (\) in front of the \*. This is necessary because Git does its own filename expansion in addition to your shell’s filename expansion. This command removes all files that have the .log extension in the log/ directory. Or, you can do something like this:

$ git rm \\*~

This command removes all files whose names end with a ~.

### **Moving Files**

Unlike many other VCSs, Git doesn’t explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact — we’ll deal with detecting file movement a bit later.

Thus it’s a bit confusing that Git has a mv command. If you want to rename a file in Git, you can run something like:

$ git mv file\_from file\_to

and it works fine. In fact, if you run something like this and look at the status, you’ll see that Git considers it a renamed file:

$ git mv README.md README

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

However, this is equivalent to running something like this:

$ mv README.md README

$ git rm README.md

$ git add README

Git figures out that it’s a rename implicitly, so it doesn’t matter if you rename a file that way or with the mv command. The only real difference is that git mv is one command instead of three — it’s a convenience function. More importantly, you can use any tool you like to rename a file, and address the add/rm later, before you commit.

* Viewing the Commit History

When you run git log in this project, you should get output that looks something like this:

$ git log

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

Remove unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

Initial commit

By default, with no arguments, git log lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author’s name and email, the date written, and the commit message.

A huge number and variety of options to the git log command are available to show you exactly what you’re looking for. Here, we’ll show you some of the most popular.

One of the more helpful options is -p or --patch, which shows the difference (the **patch** output) introduced in each commit. You can also limit the number of log entries displayed, such as using -2 to show only the last two entries.

$ git log -p -2

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -5,7 +5,7 @@ require 'rake/gempackagetask'

spec = Gem::Specification.new do |s|

s.platform = Gem::Platform::RUBY

s.name = "simplegit"

- s.version = "0.1.0"

+ s.version = "0.1.1"

s.author = "Scott Chacon"

s.email = "schacon@gee-mail.com"

s.summary = "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

Remove unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index a0a60ae..47c6340 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -18,8 +18,3 @@ class SimpleGit

end

end

-

-if $0 == \_\_FILE\_\_

- git = SimpleGit.new

- puts git.show

-end

This option displays the same information but with a diff directly following each entry. This is very helpful for code review or to quickly browse what happened during a series of commits that a collaborator has added. You can also use a series of summarizing options with git log. For example, if you want to see some abbreviated stats for each commit, you can use the --stat option:

$ git log --stat

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

Rakefile | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

Remove unnecessary test

lib/simplegit.rb | 5 -----

1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

Initial commit

README | 6 ++++++

Rakefile | 23 +++++++++++++++++++++++

lib/simplegit.rb | 25 +++++++++++++++++++++++++

3 files changed, 54 insertions(+)

As you can see, the --stat option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

Another really useful option is --pretty. This option changes the log output to formats other than the default. A few prebuilt option values are available for you to use. The oneline value for this option prints each commit on a single line, which is useful if you’re looking at a lot of commits. In addition, the short, full, and fuller values show the output in roughly the same format but with less or more information, respectively:

$ git log --pretty=oneline

ca82a6dff817ec66f44342007202690a93763949 Change version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Remove unnecessary test

a11bef06a3f659402fe7563abf99ad00de2209e6 Initial commit

The most interesting option value is format, which allows you to specify your own log output format. This is especially useful when you’re generating output for machine parsing — because you specify the format explicitly, you know it won’t change with updates to Git:

$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 6 years ago : Change version number

085bb3b - Scott Chacon, 6 years ago : Remove unnecessary test

a11bef0 - Scott Chacon, 6 years ago : Initial commit

[Useful specifiers for git log --pretty=format](https://git-scm.com/book/en/v2/ch00/pretty_format) lists some of the more useful specifiers that format takes.

| **Specifier** | **Description of Output** |
| --- | --- |
| %H | Commit hash |
| %h | Abbreviated commit hash |
| %T | Tree hash |
| %t | Abbreviated tree hash |
| %P | Parent hashes |
| %p | Abbreviated parent hashes |
| %an | Author name |
| %ae | Author email |
| %ad | Author date (format respects the --date=option) |
| %ar | Author date, relative |
| %cn | Committer name |
| %ce | Committer email |
| %cd | Committer date |
| %cr | Committer date, relative |
| %s | Subject |

You may be wondering what the difference is between **author** and **committer**. The author is the person who originally wrote the work, whereas the committer is the person who last applied the work. So, if you send in a patch to a project and one of the core members applies the patch, both of you get credit — you as the author, and the core member as the committer. We’ll cover this distinction a bit more in [Distributed Git](https://git-scm.com/book/en/v2/ch00/ch05-distributed-git).

The oneline and format option values are particularly useful with another log option called --graph. This option adds a nice little ASCII graph showing your branch and merge history:

$ git log --pretty=format:"%h %s" --graph

\* 2d3acf9 Ignore errors from SIGCHLD on trap

\* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit

|\

| \* 420eac9 Add method for getting the current branch

\* | 30e367c Timeout code and tests

\* | 5a09431 Add timeout protection to grit

\* | e1193f8 Support for heads with slashes in them

|/

\* d6016bc Require time for xmlschema

\* 11d191e Merge branch 'defunkt' into local

This type of output will become more interesting as we go through branching and merging in the next chapter.

### 

### **Limiting Log Output**

In addition to output-formatting options, git log takes a number of useful limiting options; that is, options that let you show only a subset of commits. You’ve seen one such option already — the -2 option, which displays only the last two commits. In fact, you can do -<n>, where n is any integer to show the last n commits. In reality, you’re unlikely to use that often, because Git by default pipes all output through a pager so you see only one page of log output at a time.

However, the time-limiting options such as --since and --until are very useful. For example, this command gets the list of commits made in the last two weeks:

$ git log --since=2.weeks

This command works with lots of formats — you can specify a specific date like "2008-01-15", or a relative date such as "2 years 1 day 3 minutes ago".

You can also filter the list to commits that match some search criteria. The --author option allows you to filter on a specific author, and the --grep option lets you search for keywords in the commit messages.

Another really helpful filter is the -S option (colloquially referred to as Git’s “pickaxe” option), which takes a string and shows only those commits that changed the number of occurrences of that string. For instance, if you wanted to find the last commit that added or removed a reference to a specific function, you could call:

$ git log -S function\_name

The last really useful option to pass to git log as a filter is a path. If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files. This is always the last option and is generally preceded by double dashes (--) to separate the paths from the options:

$ git log -- path/to/file

* Undoing Things

### **Unstaging a Staged File**

The next two sections demonstrate how to work with your staging area and working directory changes. The nice part is that the command you use to determine the state of those two areas also reminds you how to undo changes to them. For example, let’s say you’ve changed two files and want to commit them as two separate changes, but you accidentally type git add \* and stage them both. How can you unstage one of the two? The git status command reminds you:

$ git add \*

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

modified: CONTRIBUTING.md

Right below the “Changes to be committed” text, it says use git reset HEAD <file>…​ to unstage. So, let’s use that advice to unstage the CONTRIBUTING.md file:

$ git reset HEAD CONTRIBUTING.md

Unstaged changes after reset:

M CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

### **Unmodifying a Modified File**

What if you realize that you don’t want to keep your changes to the CONTRIBUTING.md file? How can you easily unmodify it — revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)? Luckily, git status tells you how to do that, too. In the last example output, the unstaged area looks like this:

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

It tells you pretty explicitly how to discard the changes you’ve made. Let’s do what it says:

$ git checkout -- CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

You can see that the changes have been reverted.

### **Undoing things with git restore**

Git version 2.23.0 introduced a new command: git restore. It’s basically an alternative to git reset which we just covered. From Git version 2.23.0 onwards, Git will use git restore instead of git reset for many undo operations.

Let’s retrace our steps, and undo things with git restore instead of git reset.

#### **Unstaging a Staged File with git restore**

The next two sections demonstrate how to work with your staging area and working directory changes with git restore. The nice part is that the command you use to determine the state of those two areas also reminds you how to undo changes to them. For example, let’s say you’ve changed two files and want to commit them as two separate changes, but you accidentally type git add \* and stage them both. How can you unstage one of the two? The git status command reminds you:

$ git add \*

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: CONTRIBUTING.md

renamed: README.md -> README

Right below the “Changes to be committed” text, it says use git restore --staged <file>…​ to unstage. So, let’s use that advice to unstage the CONTRIBUTING.md file:

$ git restore --staged CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

renamed: README.md -> README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

The CONTRIBUTING.md file is modified but once again unstaged.

#### **Unmodifying a Modified File with git restore**

What if you realize that you don’t want to keep your changes to the CONTRIBUTING.md file? How can you easily unmodify it — revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)? Luckily, git status tells you how to do that, too. In the last example output, the unstaged area looks like this:

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

It tells you pretty explicitly how to discard the changes you’ve made. Let’s do what it says:

$ git restore CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

renamed: README.md -> README

* Tagging

Like most VCSs, Git has the ability to tag specific points in a repository’s history as being important. Typically, people use this functionality to mark release points (v1.0, v2.0 and so on). In this section, you’ll learn how to list existing tags, how to create and delete tags, and what the different types of tags are.

### **Listing Your Tags**

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

$ git tag

v1.0

v2.0

This command lists the tags in alphabetical order; the order in which they are displayed has no real importance.

You can also search for tags that match a particular pattern. The Git source repo, for instance, contains more than 500 tags. If you’re interested only in looking at the 1.8.5 series, you can run this:

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

v1.8.5

v1.8.5-rc0

v1.8.5-rc1

v1.8.5-rc2

v1.8.5-rc3

v1.8.5.1

v1.8.5.2

v1.8.5.3

v1.8.5.4

v1.8.5.5

| **Note** | **Listing tag wildcards requires -l or --list option**  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. They’re checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It’s generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don’t want to keep the other information, lightweight tags are available too.

### **Annotated Tags**

Creating an annotated tag in Git is simple. The easiest way is to specify -a when you run the tag command:

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

$ git tag

v0.1

v1.3

v1.4

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

tag v1.4

Tagger: Ben Straub <ben@straub.cc>

Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

That shows the tagger information, the date the commit was tagged, and the annotation message before showing the commit information.

### **Lightweight Tags**

Another way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file — no other information is kept. To create a lightweight tag, don’t supply any of the -a, -s, or -m options, just provide a tag name:

$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

This time, if you run git show on the tag, you don’t see the extra tag information. The command just shows the commit:

$ git show v1.4-lw

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

### **Tagging Later**

You can also tag commits after you’ve moved past them. Suppose your commit history looks like this:

$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 Create write support

0d52aaab4479697da7686c15f77a3d64d9165190 One more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc Add commit function

4682c3261057305bdd616e23b64b0857d832627b Add todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support

9fceb02d0ae598e95dc970b74767f19372d61af8 Update rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme

Now, suppose you forgot to tag the project at v1.2, which was at the “Update rakefile” commit. You can add it after the fact. 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

You can see that you’ve tagged the commit:

$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5

$ git show v1.2

tag v1.2

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date: Sun Apr 27 20:43:35 2008 -0700

Update rakefile

...

### **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

Counting objects: 14, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (12/12), done.

Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.

Total 14 (delta 3), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.5 -> v1.5

If you have a lot of tags that you want to push up at once, 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

Counting objects: 1, done.

Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.

Total 1 (delta 0), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.4 -> v1.4

\* [new tag] v1.4-lw -> v1.4-lw

Now, when someone else clones or pulls from your repository, they will get all your tags as well.

| **Note** | **git push pushes both types of tags**  git push <remote> --tags will push both lightweight and annotated tags. There is currently no option to push only lightweight tags, but if you use git push <remote> --follow-tags only annotated tags will be pushed to the remote. |
| --- | --- |

### **Deleting Tags**

To delete a tag on your local repository, you can use git tag -d <tagname>. For example, we could remove our lightweight tag above as follows:

$ git tag -d v1.4-lw

### Git Branching

* Branching and Merging

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author’s name and email address, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

To visualize this, let’s assume that you have a directory containing three files, and you stage them all and commit. Staging the files computes a checksum for each one (the SHA-1 hash we mentioned in [What is Git?](https://git-scm.com/book/en/v2/ch00/what_is_git_section)), stores that version of the file in the Git repository (Git refers to them as **blobs**), and adds that checksum to the staging area:

$ git add README test.rb LICENSE

$ git commit -m 'Initial commit'

When you create the commit by running git commit, Git checksums each subdirectory (in this case, just the root project directory) and stores them as a tree object in the Git repository. Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

Your Git repository now contains five objects: three **blobs** (each representing the contents of one of the three files), one **tree** that lists the contents of the directory and specifies which file names are stored as which blobs, and one **commit** with the pointer to that root tree and all the commit metadata.



Figure 9. A commit and its tree

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.



Figure 10. Commits and their parents

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you’re given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

| **Note** | The “master” branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the git init command creates it by default and most people don’t bother to change it. |
| --- | --- |



Figure 11. A branch and its commit history

### **Creating a New Branch**

What happens when you create a new branch? Well, doing so creates a new pointer for you to move around. Let’s say you want to create a new branch called testing. You do this with the git branch command:

$ git branch testing

This creates a new pointer to the same commit you’re currently on.



Figure 12. Two branches pointing into the same series of commits

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. In this case, you’re still on master. The git branch command only **created** a new branch — it didn’t switch to that branch.



### **Basic Branching**

First, let’s say you’re working on your project and have a couple of commits already on the master branch.



Figure 18. A simple commit history

You’ve decided that you’re going to work on issue #53 in whatever issue-tracking system your company uses. To create a new branch and switch to it at the same time, you can run the git checkout command with the -b switch:

$ git checkout -b iss53

Switched to a new branch "iss53"

This is shorthand for:

$ git branch iss53

$ git checkout iss53



Figure 19. Creating a new branch pointer

You work on your website and do some commits. Doing so moves the iss53 branch forward, because you have it checked out (that is, your HEAD is pointing to it):

$ vim index.html

$ git commit -a -m 'Create new footer [issue 53]'



Figure 20. The iss53 branch has moved forward with your work

Now you get the call that there is an issue with the website, and you need to fix it immediately. With Git, you don’t have to deploy your fix along with the iss53 changes you’ve made, and you don’t have to put a lot of effort into reverting those changes before you can work on applying your fix to what is in production. All you have to do is switch back to your master branch.

However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you’re checking out, Git won’t let you switch branches. It’s best to have a clean working state when you switch branches. There are ways to get around this (namely, stashing and commit amending) that we’ll cover later on, in [Stashing and Cleaning](https://git-scm.com/book/en/v2/ch00/_git_stashing). For now, let’s assume you’ve committed all your changes, so you can switch back to your master branch:

$ git checkout master

Switched to branch 'master'

At this point, your project working directory is exactly the way it was before you started working on issue #53, and you can concentrate on your hotfix. This is an important point to remember: when you switch branches, Git resets your working directory to look like it did the last time you committed on that branch. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

Next, you have a hotfix to make. Let’s create a hotfix branch on which to work until it’s completed:

$ git checkout -b hotfix

Switched to a new branch 'hotfix'

$ vim index.html

$ git commit -a -m 'Fix broken email address'

[hotfix 1fb7853] Fix broken email address

1 file changed, 2 insertions(+)



Figure 21. Hotfix branch based on master

You can run your tests, make sure the hotfix is what you want, and finally merge the hotfix branch back into your master branch to deploy to production. You do this with the git merge command:

$ git checkout master

$ git merge hotfix

Updating f42c576..3a0874c

Fast-forward

index.html | 2 ++

1 file changed, 2 insertions(+)

You’ll notice the phrase “fast-forward” in that merge. Because the commit C4 pointed to by the branch hotfix you merged in was directly ahead of the commit C2 you’re on, Git simply moves the pointer forward. To phrase that another way, when you try to merge one commit with a commit that can be reached by following the first commit’s history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a “fast-forward.”

Your change is now in the snapshot of the commit pointed to by the master branch, and you can deploy the fix.



Figure 22. master is fast-forwarded to hotfix

After your super-important fix is deployed, you’re ready to switch back to the work you were doing before you were interrupted. However, first you’ll delete the hotfix branch, because you no longer need it — the master branch points at the same place. You can delete it with the -d option to git branch:

$ git branch -d hotfix

Deleted branch hotfix (3a0874c).

Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.

$ git checkout iss53

Switched to branch "iss53"

$ vim index.html

$ git commit -a -m 'Finish the new footer [issue 53]'

[iss53 ad82d7a] Finish the new footer [issue 53]

1 file changed, 1 insertion(+)



Figure 23. Work continues on iss53

It’s worth noting here that the work you did in your hotfix branch is not contained in the files in your iss53 branch. If you need to pull it in, you can merge your master branch into your iss53 branch by running git merge master, or you can wait to integrate those changes until you decide to pull the iss53 branch back into master later.

### **Basic Merging**

Suppose you’ve decided that your issue #53 work is complete and ready to be merged into your master branch. In order to do that, you’ll merge your iss53 branch into master, much like you merged your hotfix branch earlier. All you have to do is check out the branch you wish to merge into and then run the git merge command:

$ git checkout master

Switched to branch 'master'

$ git merge iss53

Merge made by the 'recursive' strategy.

index.html | 1 +

1 file changed, 1 insertion(+)

This looks a bit different than the hotfix merge you did earlier. In this case, your development history has diverged from some older point. Because the commit on the branch you’re on isn’t a direct ancestor of the branch you’re merging in, Git has to do some work. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.



Figure 24. Three snapshots used in a typical merge

Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.

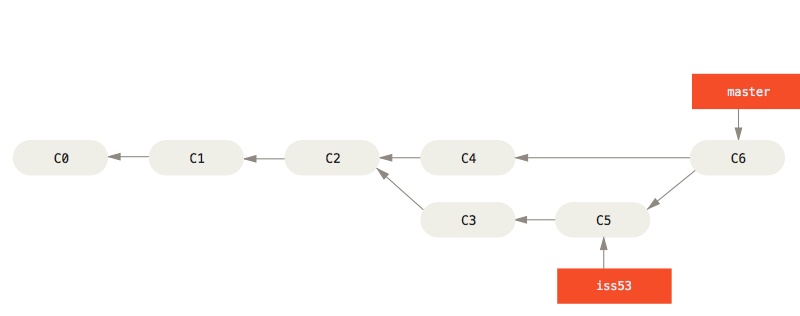


Figure 25. A merge commit

Now that your work is merged in, you have no further need for the iss53 branch. You can close the issue in your issue-tracking system, and delete the branch:

$ git branch -d iss53

### **Basic Merge Conflicts**

Occasionally, this process doesn’t go smoothly. If you changed the same part of the same file differently in the two branches you’re merging, Git won’t be able to merge them cleanly. If your fix for issue #53 modified the same part of a file as the hotfix branch, you’ll get a merge conflict that looks something like this:

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.

Git hasn’t automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run git status:

$ git status

On branch master

You have unmerged paths.

(fix conflicts and run "git commit")

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified: index.html

no changes added to commit (use "git add" and/or "git commit -a")

Anything that has merge conflicts and hasn’t been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>

>>>>>>> iss53:index.html

This means the version in HEAD (your master branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the =======), while the version in your iss53 branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this:

<div id="footer">

please contact us at email.support@github.com

</div>

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:

$ git mergetool

This message is displayed because 'merge.tool' is not configured.

See 'git mergetool --tool-help' or 'git help config' for more details.

'git mergetool' will now attempt to use one of the following tools:

opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge

Merging:

index.html

Normal merge conflict for 'index.html':

{local}: modified file

{remote}: modified file

Hit return to start merge resolution tool (opendiff):

If you want to use a merge tool other than the default (Git chose opendiff in this case because the command was run on a Mac), you can see all the supported tools listed at the top after “one of the following tools.” Just type the name of the tool you’d rather use.

* Branch Management

Now that you’ve created, merged, and deleted some branches, let’s look at some branch-management tools that will come in handy when you begin using branches all the time.

The git branch command does more than just create and delete branches. If you run it with no arguments, you get a simple listing of your current branches:

$ git branch

iss53

\* master

testing

Notice the \* character that prefixes the master branch: it indicates the branch that you currently have checked out (i.e., the branch that HEAD points to). This means that if you commit at this point, the master branch will be moved forward with your new work. To see the last commit on each branch, you can run git branch -v:

$ git branch -v

iss53 93b412c Fix javascript issue

\* master 7a98805 Merge branch 'iss53'

testing 782fd34 Add scott to the author list in the readme

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

iss53

\* master

Because you already merged in iss53 earlier, you see it in your list. Branches on this list without the \* in front of them are generally fine to delete with git branch -d; you’ve already incorporated their work into another branch, so you’re not going to lose anything.

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

$ git branch --no-merged

testing

This shows your other branch. Because it contains work that isn’t merged in yet, trying to delete it with git branch -d will fail:

$ git branch -d testing

error: The branch 'testing' is not fully merged.

If you are sure you want to delete it, run 'git branch -D testing'.

If you really do want to delete the branch and lose that work, you can force it with -D, as the helpful message points out.

| **Tip** | 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  topicA  featureB |
| --- | --- |

### **Changing a branch name**

| **Caution** | Do not rename branches that are still in use by other collaborators. Do not rename a branch like master/main/mainline without having read the section "Changing the master branch name". |
| --- | --- |

Suppose you have a branch that is called bad-branch-name and you want to change it to corrected-branch-name, while keeping all history. You also want to change the branch name on the remote (GitHub, GitLab, other server). How do you do this?

Rename the branch locally with the git branch --move command:

$ git branch --move bad-branch-name corrected-branch-name

This replaces your bad-branch-name with corrected-branch-name, but this change is only local for now. To let others see the corrected branch on the remote, push it:

$ git push --set-upstream origin corrected-branch-name

Now we’ll take a brief look at where we are now:

$ git branch --all

\* corrected-branch-name

main

remotes/origin/bad-branch-name

remotes/origin/corrected-branch-name

remotes/origin/main

Notice that you’re on the branch corrected-branch-name and it’s available on the remote. However, the branch with the bad name is also still present there but you can delete it by executing the following command:

$ git push origin --delete bad-branch-name

Now the bad branch name is fully replaced with the corrected branch name.

#### **Changing the master branch name**

| **Warning** | Changing the name of a branch like master/main/mainline/default will break the integrations, services, helper utilities and build/release scripts that your repository uses. Before you do this, make sure you consult with your collaborators. Also, make sure you do a thorough search through your repo and update any references to the old branch name in your code and scripts. |
| --- | --- |

Rename your local master branch into main with the following command:

$ git branch --move master main

There’s no local master branch anymore, because it’s renamed to the main branch.

To let others see the new main branch, you need to push it to the remote. This makes the renamed branch available on the remote.

$ git push --set-upstream origin main

Now we end up with the following state:

$ git branch --all

\* main

remotes/origin/HEAD -> origin/master

remotes/origin/main

remotes/origin/master

Your local master branch is gone, as it’s replaced with the main branch. The main branch is present on the remote. However, the old master branch is still present on the remote. Other collaborators will continue to use the master branch as the base of their work, until you make some further changes.

Now you have a few more tasks in front of you to complete the transition:

* Any projects that depend on this one will need to update their code and/or configuration.
* Update any test-runner configuration files.
* Adjust build and release scripts.
* Redirect settings on your repo host for things like the repo’s default branch, merge rules, and other things that match branch names.
* Update references to the old branch in documentation.
* Close or merge any pull requests that target the old branch.

After you’ve done all these tasks, and are certain the main branch performs just as the master branch, you can delete the master branch:

$ git push origin --delete master

* Branching Workflows

### **Long-Running Branches**

Because Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do. This means you can have several branches that are always open and that you use for different stages of your development cycle; you can merge regularly from some of them into others.

Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their master branch — possibly only code that has been or will be released. They have another parallel branch named develop or next that they work from or use to test stability — it isn’t necessarily always stable, but whenever it gets to a stable state, it can be merged into master. It’s used to pull in topic branches (short-lived branches, like your earlier iss53 branch) when they’re ready, to make sure they pass all the tests and don’t introduce bugs.

In reality, we’re talking about pointers moving up the line of commits you’re making. The stable branches are farther down the line in your commit history, and the bleeding-edge branches are farther up the history.



Figure 26. A linear view of progressive-stability branching

It’s generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they’re fully tested.



Figure 27. A “silo” view of progressive-stability branching

You can keep doing this for several levels of stability. Some larger projects also have a proposed or pu (proposed updates) branch that has integrated branches that may not be ready to go into the next or master branch. The idea is that your branches are at various levels of stability; when they reach a more stable level, they’re merged into the branch above them. Again, having multiple long-running branches isn’t necessary, but it’s often helpful, especially when you’re dealing with very large or complex projects.

### **Topic Branches**

Topic branches, however, are useful in projects of any size. A topic branch is a short-lived branch that you create and use for a single particular feature or related work. This is something you’ve likely never done with a VCS before because it’s generally too expensive to create and merge branches. But in Git it’s common to create, work on, merge, and delete branches several times a day.

You saw this in the last section with the iss53 and hotfix branches you created. You did a few commits on them and deleted them directly after merging them into your main branch. This technique allows you to context-switch quickly and completely — because your work is separated into silos where all the changes in that branch have to do with that topic, it’s easier to see what has happened during code review and such. You can keep the changes there for minutes, days, or months, and merge them in when they’re ready, regardless of the order in which they were created or worked on.

Consider an example of doing some work (on master), branching off for an issue (iss91), working on it for a bit, branching off the second branch to try another way of handling the same thing (iss91v2), going back to your master branch and working there for a while, and then branching off there to do some work that you’re not sure is a good idea (dumbidea branch). Your commit history will look something like this:



Figure 28. Multiple topic branches

Now, let’s say you decide you like the second solution to your issue best (iss91v2); and you showed the dumbidea branch to your coworkers, and it turns out to be genius. You can throw away the original iss91 branch (losing commits C5 and C6) and merge in the other two. Your history then looks like this:



Figure 29. History after merging dumbidea and iss91v2

We will go into more detail about the various possible workflows for your Git project in [Distributed Git](https://git-scm.com/book/en/v2/ch00/ch05-distributed-git), so before you decide which branching scheme your next project will use, be sure to read that chapter.

* Remote Branches

Remote references are references (pointers) in your remote repositories, including branches, tags, and so on. 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. Nevertheless, a more common way is to take advantage of remote-tracking branches.

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 branch names take the form <remote>/<branch>. For instance, if you wanted to see what the master branch on your origin remote looked like as of the last time you communicated with it, you would check the origin/master branch. If you were working on an issue with a partner and they pushed up an iss53 branch, you might have your own local iss53 branch, but the branch on the server would be represented by the remote-tracking branch origin/iss53.

This may be a bit confusing, so let’s look at an example. Let’s say you have a Git server on your network at git.ourcompany.com. If you clone from this, Git’s clone command automatically names it origin for you, pulls down all its data, creates a pointer to where its master branch is, and names it origin/master locally. Git also gives you your own local master branch starting at the same place as origin’s master branch, so you have something to work from.

* Rebasing

In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase. In this section you’ll learn what rebasing is, how to do it, why it’s a pretty amazing tool, and in what cases you won’t want to use it.

### **The Basic Rebase**

If you go back to an earlier example from [Basic Merging](https://git-scm.com/book/en/v2/ch00/_basic_merging), you can see that you diverged your work and made commits on two different branches.



Figure 35. Simple divergent history

The easiest way to integrate the branches, as we’ve already covered, is the merge command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit).



Figure 36. Merging to integrate diverged work history

However, there is another way: you can take the patch of the change that was introduced in C4 and reapply it on top of C3. In Git, this is called **rebasing**. With the rebase command, you can take all the changes that were committed on one branch and replay them on a different branch.

For this example, you would check out the experiment branch, and then rebase it onto the master branch as follows:

$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: added staged command

This operation 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.



Figure 37. Rebasing the change introduced in C4 onto C3

At this point, you can go back to the master branch and do a fast-forward merge.

$ git checkout master

$ git merge experiment



Figure 38. Fast-forwarding the master branch

Now, the snapshot pointed to by C4' is exactly the same as the one that was pointed to by C5 in [the merge example](https://git-scm.com/book/en/v2/ch00/rebasing-merging-example). 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.

Often, you’ll do this to make sure your commits apply cleanly on a remote branch — perhaps in a project to which you’re trying to contribute but that you don’t maintain. In this case, you’d do your work in a branch and then rebase your work onto origin/master when you were ready to submit your patches to the main project. That way, the maintainer doesn’t have to do any integration work — just a fast-forward or a clean apply.

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**

You can also have your rebase replay on something other than the rebase target branch. Take a history like [A history with a topic branch off another topic branch](https://git-scm.com/book/en/v2/ch00/rbdiag_e), for example. You branched a topic branch (server) to add some server-side functionality to your project, and made a commit. Then, you branched off that to make the client-side changes (client) and committed a few times. Finally, you went back to your server branch and did a few more commits.

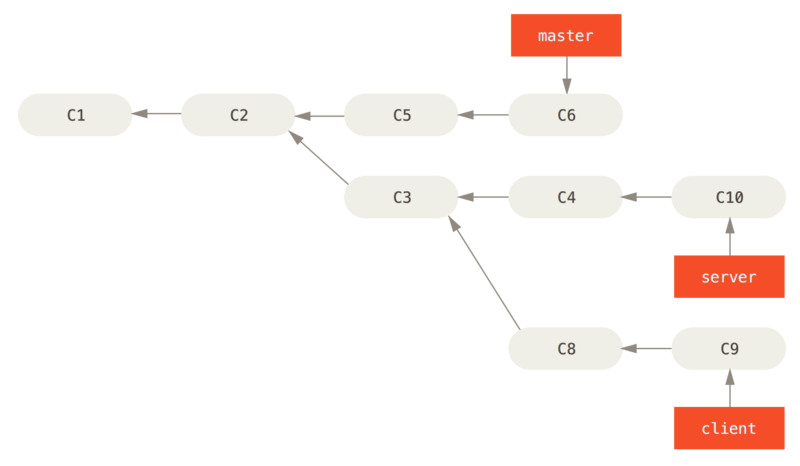


Figure 39. A history with a topic branch off another topic branch

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.



Figure 40. Rebasing a topic branch off another topic branch

Now you can fast-forward your master branch (see [Fast-forwarding your master branch to include the client branch changes](https://git-scm.com/book/en/v2/ch00/rbdiag_g)):

$ git checkout master

$ git merge client



Figure 41. Fast-forwarding your master branch to include the client branch changes

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

This replays your server work on top of your master work, as shown in [Rebasing your server branch on top of your master branch](https://git-scm.com/book/en/v2/ch00/rbdiag_h).



Figure 42. Rebasing your server branch on top of your master branch

Then, you can fast-forward the base branch (master):

$ git checkout master

$ git merge server

You can remove the client and server branches because all the work is integrated and you don’t need them anymore, leaving your history for this entire process looking like [Final commit history](https://git-scm.com/book/en/v2/ch00/rbdiag_i):

$ git branch -d client

$ git branch -d server



Figure 43. Final commit history

### **The Perils of Rebasing**

Ahh, but the bliss of rebasing isn’t without its drawbacks, which can be summed up in a single line:

**Do not rebase commits that exist outside your repository and that people may have based work on.**

If you follow that guideline, you’ll be fine. If you don’t, people will hate you, and you’ll be scorned by friends and family.

When you rebase stuff, you’re abandoning existing commits and creating new ones that are similar but different. If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with git rebase and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

Let’s look at an example of how rebasing work that you’ve made public can cause problems. Suppose you clone from a central server and then do some work off that. Your commit history looks like this:



Figure 44. Clone a repository, and base some work on it

Now, someone else does more work that includes a merge, and pushes that work to the central server. You fetch it and merge the new remote branch into your work, making your history look something like this:



Figure 45. Fetch more commits, and merge them into your work

Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a git push --force to overwrite the history on the server. You then fetch from that server, bringing down the new commits.



Figure 46. Someone pushes rebased commits, abandoning commits you’ve based your work on

Now you’re both in a pickle. If you do a git pull, you’ll create a merge commit which includes both lines of history, and your repository will look like this:



Figure 47. You merge in the same work again into a new merge commit

If you run a git log when your history looks like this, you’ll see two commits that have the same author, date, and message, which will be confusing. Furthermore, if you push this history back up to the server, you’ll reintroduce all those rebased commits to the central server, which can further confuse people. It’s pretty safe to assume that the other developer doesn’t want C4 and C6 to be in the history; that’s why they rebased in the first place.

### **Rebase When You Rebase**

If you **do** find yourself in a situation like this, Git has some further magic that might help you out. If someone on your team force pushes changes that overwrite work that you’ve based work on, your challenge is to figure out what is yours and what they’ve rewritten.

It turns out that in addition to the commit SHA-1 checksum, Git also calculates a checksum that is based just on the patch introduced with the commit. This is called a “patch-id”.

If you pull down work that was rewritten and rebase it on top of the new commits from your partner, Git can often successfully figure out what is uniquely yours and apply them back on top of the new branch.

For instance, in the previous scenario, if instead of doing a merge when we’re at [Someone pushes rebased commits, abandoning commits you’ve based your work on](https://git-scm.com/book/en/v2/ch00/_pre_merge_rebase_work) we run git rebase teamone/master, Git will:

* Determine what work is unique to our branch (C2, C3, C4, C6, C7)
* Determine which are not merge commits (C2, C3, C4)
* Determine which have not been rewritten into the target branch (just C2 and C3, since C4 is the same patch as C4')
* Apply those commits to the top of teamone/master

So instead of the result we see in [You merge in the same work again into a new merge commit](https://git-scm.com/book/en/v2/ch00/_merge_rebase_work), we would end up with something more like [Rebase on top of force-pushed rebase work](https://git-scm.com/book/en/v2/ch00/_rebase_rebase_work).



Figure 48. Rebase on top of force-pushed rebase work

This only works if C4 and C4' that your partner made are almost exactly the same patch. Otherwise the rebase won’t be able to tell that it’s a duplicate and will add another C4-like patch (which will probably fail to apply cleanly, since the changes would already be at least somewhat there).

You can also simplify this by running a git pull --rebase instead of a normal git pull. Or you could do it manually with a git fetch followed by a git rebase teamone/master in this case.

If you are using git pull and want to make --rebase the default, you can set the pull.rebase config value with something like git config --global pull.rebase true.

If you only ever rebase commits that have never left your own computer, you’ll be just fine. If you rebase commits that have been pushed, but that no one else has based commits from, you’ll also be fine. If you rebase commits that have already been pushed publicly, and people may have based work on those commits, then you may be in for some frustrating trouble, and the scorn of your teammates.

If you or a partner does find it necessary at some point, make sure everyone knows to run git pull --rebase to try to make the pain after it happens a little bit simpler.

### **Git on the Server**

* The Protocols

Git can use four distinct protocols to transfer data: Local, HTTP, Secure Shell (SSH) and Git. Here we’ll discuss what they are and in what basic circumstances you would want (or not want) to use them.

### **Local Protocol**

The most basic is the **Local protocol**, in which the remote repository is in another directory on the same host. This is often used if everyone on your team has access to a shared filesystem such as an [NFS](https://en.wikipedia.org/wiki/Network_File_System) mount, or in the less likely case that everyone logs in to the same computer. The latter wouldn’t be ideal, because all your code repository instances would reside on the same computer, making a catastrophic loss much more likely.

If you have a shared mounted filesystem, then you can clone, push to, and pull from a local file-based repository. To clone a repository like this, or to add one as a remote to an existing project, use the path to the repository as the URL. For example, to clone a local repository, you can run something like this:

$ git clone /srv/git/project.git

Or you can do this:

$ git clone file:///srv/git/project.git

Git operates slightly differently if you explicitly specify file:// at the beginning of the URL. If you just specify the path, Git tries to use hardlinks or directly copy the files it needs. If you specify file://, Git fires up the processes that it normally uses to transfer data over a network, which is generally much less efficient. The main reason to specify the file:// prefix is if you want a clean copy of the repository with extraneous references or objects left out — generally after an import from another VCS or something similar (see [Git Internals](https://git-scm.com/book/en/v2/ch00/ch10-git-internals) for maintenance tasks). We’ll use the normal path here because doing so is almost always faster.

To add a local repository to an existing Git project, you can run something like this:

$ git remote add local\_proj /srv/git/project.git

Then, you can push to and pull from that remote via your new remote name local\_proj as though you were doing so over a network.

#### **The Pros**

The pros of file-based repositories are that they’re simple and they use existing file permissions and network access. If you already have a shared filesystem to which your whole team has access, setting up a repository is very easy. You stick the bare repository copy somewhere everyone has shared access to and set the read/write permissions as you would for any other shared directory. We’ll discuss how to export a bare repository copy for this purpose in [Getting Git on a Server](https://git-scm.com/book/en/v2/ch00/_getting_git_on_a_server).

This is also a nice option for quickly grabbing work from someone else’s working repository. If you and a co-worker are working on the same project and they want you to check something out, running a command like git pull /home/john/project is often easier than them pushing to a remote server and you subsequently fetching from it.

#### **The Cons**

The cons of this method are that shared access is generally more difficult to set up and reach from multiple locations than basic network access. If you want to push from your laptop when you’re at home, you have to mount the remote disk, which can be difficult and slow compared to network-based access.

It’s important to mention that this isn’t necessarily the fastest option if you’re using a shared mount of some kind. A local repository is fast only if you have fast access to the data. A repository on NFS is often slower than the repository over SSH on the same server, allowing Git to run off local disks on each system.

Finally, this protocol does not protect the repository against accidental damage. Every user has full shell access to the “remote” directory, and there is nothing preventing them from changing or removing internal Git files and corrupting the repository.

### **The HTTP Protocols**

Git can communicate over HTTP using two different modes. Prior to Git 1.6.6, there was only one way it could do this which was very simple and generally read-only. In version 1.6.6, a new, smarter protocol was introduced that involved Git being able to intelligently negotiate data transfer in a manner similar to how it does over SSH. In the last few years, this new HTTP protocol has become very popular since it’s simpler for the user and smarter about how it communicates. The newer version is often referred to as the **Smart** HTTP protocol and the older way as **Dumb** HTTP. We’ll cover the newer Smart HTTP protocol first.

#### **Smart HTTP**

Smart HTTP operates very similarly to the SSH or Git protocols but runs over standard HTTPS ports and can use various HTTP authentication mechanisms, meaning it’s often easier on the user than something like SSH, since you can use things like username/password authentication rather than having to set up SSH keys.

It has probably become the most popular way to use Git now, since it can be set up to both serve anonymously like the git:// protocol, and can also be pushed over with authentication and encryption like the SSH protocol. Instead of having to set up different URLs for these things, you can now use a single URL for both. If you try to push and the repository requires authentication (which it normally should), the server can prompt for a username and password. The same goes for read access.

In fact, for services like GitHub, the URL you use to view the repository online (for example, <https://github.com/schacon/simplegit>) is the same URL you can use to clone and, if you have access, push over.

#### **Dumb HTTP**

If the server does not respond with a Git HTTP smart service, the Git client will try to fall back to the simpler **Dumb** HTTP protocol. The Dumb protocol expects the bare Git repository to be served like normal files from the web server. The beauty of Dumb HTTP is the simplicity of setting it up. Basically, all you have to do is put a bare Git repository under your HTTP document root and set up a specific post-update hook, and you’re done (See [Git Hooks](https://git-scm.com/book/en/v2/ch00/_git_hooks)). At that point, anyone who can access the web server under which you put the repository can also clone your repository. To allow read access to your repository over HTTP, do something like this:

$ cd /var/www/htdocs/

$ git clone --bare /path/to/git\_project gitproject.git

$ cd gitproject.git

$ mv hooks/post-update.sample hooks/post-update

$ chmod a+x hooks/post-update

That’s all. The post-update hook that comes with Git by default runs the appropriate command (git update-server-info) to make HTTP fetching and cloning work properly. This command is run when you push to this repository (over SSH perhaps); then, other people can clone via something like:

$ git clone https://example.com/gitproject.git

In this particular case, we’re using the /var/www/htdocs path that is common for Apache setups, but you can use any static web server — just put the bare repository in its path. The Git data is served as basic static files (see the [Git Internals](https://git-scm.com/book/en/v2/ch00/ch10-git-internals) chapter for details about exactly how it’s served).

Generally you would either choose to run a read/write Smart HTTP server or simply have the files accessible as read-only in the Dumb manner. It’s rare to run a mix of the two services.

#### **The Pros**

We’ll concentrate on the pros of the Smart version of the HTTP protocol.

The simplicity of having a single URL for all types of access and having the server prompt only when authentication is needed makes things very easy for the end user. Being able to authenticate with a username and password is also a big advantage over SSH, since users don’t have to generate SSH keys locally and upload their public key to the server before being able to interact with it. For less sophisticated users, or users on systems where SSH is less common, this is a major advantage in usability. It is also a very fast and efficient protocol, similar to the SSH one.

You can also serve your repositories read-only over HTTPS, which means you can encrypt the content transfer; or you can go so far as to make the clients use specific signed SSL certificates.

Another nice thing is that HTTP and HTTPS are such commonly used protocols that corporate firewalls are often set up to allow traffic through their ports.

#### **The Cons**

Git over HTTPS can be a little more tricky to set up compared to SSH on some servers. Other than that, there is very little advantage that other protocols have over Smart HTTP for serving Git content.

If you’re using HTTP for authenticated pushing, providing your credentials is sometimes more complicated than using keys over SSH. There are, however, several credential caching tools you can use, including Keychain access on macOS and Credential Manager on Windows, to make this pretty painless. Read [Credential Storage](https://git-scm.com/book/en/v2/ch00/_credential_caching) to see how to set up secure HTTP password caching on your system.

### **The SSH Protocol**

We’re using: keys, secrets, ssl: ssh, scp

A common transport protocol for Git when self-hosting is over SSH. This is because SSH access to servers is already set up in most places — and if it isn’t, it’s easy to do. SSH is also an authenticated network protocol and, because it’s ubiquitous, it’s generally easy to set up and use.

To clone a Git repository over SSH, you can specify an ssh:// URL like this:

$ git clone ssh://[user@]server/project.git

Or you can use the shorter scp-like syntax for the SSH protocol:

$ git clone [user@]server:project.git

In both cases above, if you don’t specify the optional username, Git assumes the user you’re currently logged in as.

#### **The Pros**

The pros of using SSH are many. First, SSH is relatively easy to set up — SSH daemons are commonplace, many network admins have experience with them, and many OS distributions are set up with them or have tools to manage them. Next, access over SSH is secure — all data transfer is encrypted and authenticated. Last, like the HTTPS, Git and Local protocols, SSH is efficient, making the data as compact as possible before transferring it.

#### **The Cons**

The negative aspect of SSH is that it doesn’t support anonymous access to your Git repository. If you’re using SSH, people **must** have SSH access to your machine, even in a read-only capacity, which doesn’t make SSH conducive to open source projects for which people might simply want to clone your repository to examine it. If you’re using it only within your corporate network, SSH may be the only protocol you need to deal with. If you want to allow anonymous read-only access to your projects and also want to use SSH, you’ll have to set up SSH for you to push over but something else for others to fetch from.

### **The Git Protocol**

Finally, we have the Git protocol. This is a special daemon that comes packaged with Git; it listens on a dedicated port (9418) that provides a service similar to the SSH protocol, but with absolutely no authentication. In order for a repository to be served over the Git protocol, you must create a git-daemon-export-ok file — the daemon won’t serve a repository without that file in it — but, other than that, there is no security. Either the Git repository is available for everyone to clone, or it isn’t. This means that there is generally no pushing over this protocol. You can enable push access but, given the lack of authentication, anyone on the internet who finds your project’s URL could push to that project. Suffice it to say that this is rare.

#### **The Pros**

The Git protocol is often the fastest network transfer protocol available. If you’re serving a lot of traffic for a public project or serving a very large project that doesn’t require user authentication for read access, it’s likely that you’ll want to set up a Git daemon to serve your project. It uses the same data-transfer mechanism as the SSH protocol but without the encryption and authentication overhead.

#### **The Cons**

The downside of the Git protocol is the lack of authentication. It’s generally undesirable for the Git protocol to be the only access to your project. Generally, you’ll pair it with SSH or HTTPS access for the few developers who have push (write) access and have everyone else use git:// for read-only access. It’s also probably the most difficult protocol to set up. It must run its own daemon, which requires xinetd or systemd configuration or the like, which isn’t always a walk in the park. It also requires firewall access to port 9418, which isn’t a standard port that corporate firewalls always allow. Behind big corporate firewalls, this obscure port is commonly blocked.

* Getting Git on a Server

In order to initially set up any Git server, you have to export an existing repository into a new bare repository — a repository that doesn’t contain a working directory. This is generally straightforward to do. In order to clone your repository to create a new bare repository, you run the clone command with the --bare option. By convention, bare repository directory names end with the suffix .git, like so:

$ git clone --bare my\_project my\_project.git

Cloning into bare repository 'my\_project.git'...

done.

You should now have a copy of the Git directory data in your my\_project.git directory.

This is roughly equivalent to something like:

$ cp -Rf my\_project/.git my\_project.git

There are a couple of minor differences in the configuration file but, for your purpose, this is close to the same thing. It takes the Git repository by itself, without a working directory, and creates a directory specifically for it alone.

### **Putting the Bare Repository on a Server**

Now that you have a bare copy of your repository, all you need to do is put it on a server and set up your protocols. Let’s say you’ve set up a server called git.example.com to which you have SSH access, and you want to store all your Git repositories under the /srv/git directory. Assuming that /srv/git exists on that server, you can set up your new repository by copying your bare repository over:

$ scp -r my\_project.git user@git.example.com:/srv/git

At this point, other users who have SSH-based read access to the /srv/git directory on that server can clone your repository by running:

$ git clone user@git.example.com:/srv/git/my\_project.git

If a user SSHs into a server and has write access to the /srv/git/my\_project.git directory, they will also automatically have push access.

Git will automatically add group write permissions to a repository properly if you run the git init command with the --shared option. Note that by running this command, you will not destroy any commits, refs, etc. in the process.

$ ssh user@git.example.com

$ cd /srv/git/my\_project.git

$ git init --bare --shared

You see how easy it is to take a Git repository, create a bare version, and place it on a server to which you and your collaborators have SSH access. Now you’re ready to collaborate on the same project.

It’s important to note that this is literally all you need to do to run a useful Git server to which several people have access — just add SSH-able accounts on a server, and stick a bare repository somewhere that all those users have read and write access to. You’re ready to go — nothing else needed.

In the next few sections, you’ll see how to expand to more sophisticated setups. This discussion will include not having to create user accounts for each user, adding public read access to repositories, setting up web UIs and more. However, keep in mind that to collaborate with a couple of people on a private project, all you **need** is an SSH server and a bare repository.

### **Small Setups**

If you’re a small outfit or are just trying out Git in your organization and have only a few developers, things can be simple for you. One of the most complicated aspects of setting up a Git server is user management. If you want some repositories to be read-only for certain users and read/write for others, access and permissions can be a bit more difficult to arrange.

#### **SSH Access**

If you have a server to which all your developers already have SSH access, it’s generally easiest to set up your first repository there, because you have to do almost no work (as we covered in the last section). If you want more complex access control type permissions on your repositories, you can handle them with the normal filesystem permissions of your server’s operating system.

If you want to place your repositories on a server that doesn’t have accounts for everyone on your team for whom you want to grant write access, then you must set up SSH access for them. We assume that if you have a server with which to do this, you already have an SSH server installed, and that’s how you’re accessing the server.

There are a few ways you can give access to everyone on your team. The first is to set up accounts for everybody, which is straightforward but can be cumbersome. You may not want to run adduser (or the possible alternative useradd) and have to set temporary passwords for every new user.

A second method is to create a single 'git' user account on the machine, ask every user who is to have write access to send you an SSH public key, and add that key to the ~/.ssh/authorized\_keys file of that new 'git' account. At that point, everyone will be able to access that machine via the 'git' account. This doesn’t affect the commit data in any way — the SSH user you connect as doesn’t affect the commits you’ve recorded.

Another way to do it is to have your SSH server authenticate from an LDAP server or some other centralized authentication source that you may already have set up. As long as each user can get shell access on the machine, any SSH authentication mechanism you can think of should work.

* Generating SSH Public Key

Many Git servers authenticate using SSH public keys. In order to provide a public key, each user in your system must generate one if they don’t already have one. This process is similar across all operating systems. First, you should check to make sure you don’t already have a key. By default, a user’s SSH keys are stored in that user’s ~/.ssh directory. You can easily check to see if you have a key already by going to that directory and listing the contents:

$ cd ~/.ssh

$ ls

authorized\_keys2 id\_dsa known\_hosts

config id\_dsa.pub

You’re looking for a pair of files named something like id\_dsa or id\_rsa and a matching file with a .pub extension. The .pub file is your public key, and the other file is the corresponding private key. If you don’t have these files (or you don’t even have a .ssh directory), you can create them by running a program called ssh-keygen, which is provided with the SSH package on Linux/macOS systems and comes with Git for Windows:

$ ssh-keygen -o

Generating public/private rsa key pair.

Enter file in which to save the key (/home/schacon/.ssh/id\_rsa):

Created directory '/home/schacon/.ssh'.

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /home/schacon/.ssh/id\_rsa.

Your public key has been saved in /home/schacon/.ssh/id\_rsa.pub.

The key fingerprint is:

d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local

First it confirms where you want to save the key (.ssh/id\_rsa), and then it asks twice for a passphrase, which you can leave empty if you don’t want to type a password when you use the key. However, if you do use a password, make sure to add the -o option; it saves the private key in a format that is more resistant to brute-force password cracking than is the default format. You can also use the ssh-agent tool to prevent having to enter the password each time.

Now, each user that does this has to send their public key to you or whoever is administrating the Git server (assuming you’re using an SSH server setup that requires public keys). All they have to do is copy the contents of the .pub file and email it. The public keys look something like this:

$ cat ~/.ssh/id\_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU

GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3

Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA

t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En

mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx

NrRFi9wrf+M7Q== schacon@mylaptop.local

* Setting up the Server

First, you create a git user account and a .ssh directory for that user.

$ sudo adduser git

$ su git

$ cd

$ mkdir .ssh && chmod 700 .ssh

$ touch .ssh/authorized\_keys && chmod 600 .ssh/authorized\_keys

Next, you need to add some developer SSH public keys to the authorized\_keys file for the git user. Let’s assume you have some trusted public keys and have saved them to temporary files. Again, the public keys look something like this:

$ cat /tmp/id\_rsa.john.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L

ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k

Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez

Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv

O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq

dAv8JggJICUvax2T9va5 gsg-keypair

You just append them to the git user’s authorized\_keys file in its .ssh directory:

$ cat /tmp/id\_rsa.john.pub >> ~/.ssh/authorized\_keys

$ cat /tmp/id\_rsa.josie.pub >> ~/.ssh/authorized\_keys

$ cat /tmp/id\_rsa.jessica.pub >> ~/.ssh/authorized\_keys

Now, you can set up an empty repository for them by running git init with the --bare option, which initializes the repository without a working directory:

$ cd /srv/git

$ mkdir project.git

$ cd project.git

$ git init --bare

Initialized empty Git repository in /srv/git/project.git/

Then, John, Josie, or Jessica can push the first version of their project into that repository by adding it as a remote and pushing up a branch. Note that someone must shell onto the machine and create a bare repository every time you want to add a project. Let’s use gitserver as the hostname of the server on which you’ve set up your git user and repository. If you’re running it internally, and you set up DNS for gitserver to point to that server, then you can use the commands pretty much as is (assuming that myproject is an existing project with files in it):

# on John's computer

$ cd myproject

$ git init

$ git add .

$ git commit -m 'Initial commit'

$ git remote add origin git@gitserver:/srv/git/project.git

$ git push origin master

At this point, the others can clone it down and push changes back up just as easily:

$ git clone git@gitserver:/srv/git/project.git

$ cd project

$ vim README

$ git commit -am 'Fix for README file'

$ git push origin master

With this method, you can quickly get a read/write Git server up and running for a handful of developers.

You should note that currently all these users can also log into the server and get a shell as the git user. If you want to restrict that, you will have to change the shell to something else in the /etc/passwd file.

You can easily restrict the git user account to only Git-related activities with a limited shell tool called git-shell that comes with Git. If you set this as the git user account’s login shell, then that account can’t have normal shell access to your server. To use this, specify git-shell instead of bash or csh for that account’s login shell. To do so, you must first add the full pathname of the git-shell command to /etc/shells if it’s not already there:

$ cat /etc/shells # see if git-shell is already in there. If not...

$ which git-shell # make sure git-shell is installed on your system.

$ sudo -e /etc/shells # and add the path to git-shell from last command

Now you can edit the shell for a user using chsh <username> -s <shell>:

$ sudo chsh git -s $(which git-shell)

Now, the git user can still use the SSH connection to push and pull Git repositories but can’t shell onto the machine. If you try, you’ll see a login rejection like this:

$ ssh git@gitserver

fatal: Interactive git shell is not enabled.

hint: ~/git-shell-commands should exist and have read and execute access.

Connection to gitserver closed.

At this point, users are still able to use SSH port forwarding to access any host the git server is able to reach. If you want to prevent that, you can edit the authorized\_keys file and prepend the following options to each key you’d like to restrict:

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty

The result should look like this:

$ cat ~/.ssh/authorized\_keys

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa

AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4LojG6rs6h

PB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541N

YsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9EzSdfd8AcC

IicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myivO7TCUSBd

LQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPqdAv8JggJ

ICUvax2T9va5 gsg-keypair

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa

AAAAB3NzaC1yc2EAAAADAQABAAABAQDEwENNMomTboYI+LJieaAY16qiXiH3wuvENhBG...

Now Git network commands will still work just fine but the users won’t be able to get a shell. As the output states, you can also set up a directory in the git user’s home directory that customizes the git-shell command a bit. For instance, you can restrict the Git commands that the server will accept or you can customize the message that users see if they try to SSH in like that. Run git help shell for more information on customizing the shell.

### 

### 

### 

### 

### **Distributed Git**

* Distributed Workflows

In contrast with Centralized Version Control Systems (CVCSs), the distributed nature of Git allows you to be far more flexible in how developers collaborate on projects. In centralized systems, every developer is a node working more or less equally with a central hub. In Git, however, every developer is potentially both a node and a hub; that is, every developer can both contribute code to other repositories and maintain a public repository on which others can base their work and which they can contribute to. This presents a vast range of workflow possibilities for your project and/or your team, so we’ll cover a few common paradigms that take advantage of this flexibility. We’ll go over the strengths and possible weaknesses of each design; you can choose a single one to use, or you can mix and match features from each.

### **Centralized Workflow**

In centralized systems, there is generally a single collaboration model — the centralized workflow. One central hub, or **repository**, can accept code, and everyone synchronizes their work with it. A number of developers are nodes — consumers of that hub — and synchronize with that centralized location.



Figure 53. Centralized workflow

This means that if two developers clone from the hub and both make changes, the first developer to push their changes back up can do so with no problems. The second developer must merge in the first one’s work before pushing changes up, so as not to overwrite the first developer’s changes. This concept is as true in Git as it is in Subversion (or any CVCS), and this model works perfectly well in Git.

If you are already comfortable with a centralized workflow in your company or team, you can easily continue using that workflow with Git. Simply set up a single repository, and give everyone on your team push access; Git won’t let users overwrite each other.

Say John and Jessica both start working at the same time. John finishes his change and pushes it to the server. Then Jessica tries to push her changes, but the server rejects them. She is told that she’s trying to push non-fast-forward changes and that she won’t be able to do so until she fetches and merges. This workflow is attractive to a lot of people because it’s a paradigm that many are familiar and comfortable with.

This is also not limited to small teams. With Git’s branching model, it’s possible for hundreds of developers to successfully work on a single project through dozens of branches simultaneously.

### **Integration-Manager Workflow**

Because Git allows you to have multiple remote repositories, it’s possible to have a workflow where each developer has write access to their own public repository and read access to everyone else’s. This scenario often includes a canonical repository that represents the “official” project. To contribute to that project, you create your own public clone of the project and push your changes to it. Then, you can send a request to the maintainer of the main project to pull in your changes. The maintainer can then add your repository as a remote, test your changes locally, merge them into their branch, and push back to their repository. The process works as follows (see [Integration-manager workflow](https://git-scm.com/book/en/v2/ch00/wfdiag_b)):

1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an email asking them to pull changes.
5. The maintainer adds the contributor’s repository as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.



Figure 54. Integration-manager workflow

This is a very common workflow with hub-based tools like GitHub or GitLab, where it’s easy to fork a project and push your changes into your fork for everyone to see. One of the main advantages of this approach is that you can continue to work, and the maintainer of the main repository can pull in your changes at any time. Contributors don’t have to wait for the project to incorporate their changes — each party can work at their own pace.

### **Dictator and Lieutenants Workflow**

This is a variant of a multiple-repository workflow. It’s generally used by huge projects with hundreds of collaborators; one famous example is the Linux kernel. Various integration managers are in charge of certain parts of the repository; they’re called **lieutenants**. All the lieutenants have one integration manager known as the benevolent dictator. The benevolent dictator pushes from their directory to a reference repository from which all the collaborators need to pull. The process works like this (see [Benevolent dictator workflow](https://git-scm.com/book/en/v2/ch00/wfdiag_c)):

1. Regular developers work on their topic branch and rebase their work on top of master. The master branch is that of the reference repository to which the dictator pushes.
2. Lieutenants merge the developers' topic branches into their master branch.
3. The dictator merges the lieutenants' master branches into the dictator’s master branch.
4. Finally, the dictator pushes that master branch to the reference repository so the other developers can rebase on it.



Figure 55. Benevolent dictator workflow

This kind of workflow isn’t common, but can be useful in very big projects, or in highly hierarchical environments. It allows the project leader (the dictator) to delegate much of the work and collect large subsets of code at multiple points before integrating them.

Git is the most commonly used version control system today. A Git workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner. Git workflows encourage developers and [DevOps](https://www.atlassian.com/devops/what-is-devops) teams to leverage Git effectively and consistently. Git offers a lot of flexibility in how users manage changes. Given Git's focus on flexibility, there is no standardized process on how to interact with Git. When working with a team on a Git-managed project, it’s important to make sure the team is all in agreement on how the flow of changes will be applied. To ensure the team is on the same page, an agreed-upon Git workflow should be developed or selected. There are several publicized Git workflows that may be a good fit for your team. Here, we will discuss some of these Git workflow options.

The array of possible workflows can make it hard to know where to begin when implementing Git in the workplace. This page provides a starting point by surveying the most common Git workflows for software teams.

As you read through, remember that these workflows are designed to be guidelines rather than concrete rules. We want to show you what’s possible, so you can mix and match aspects from different workflows to suit your individual needs.

## What is a successful Git workflow?

When evaluating a workflow for your team, it's most important that you consider your team’s culture. You want the workflow to enhance the effectiveness of your team and not be a burden that limits productivity. Some things to consider when evaluating a Git workflow are:

* Does this workflow scale with team size?
* Is it easy to undo mistakes and errors with this workflow?
* Does this workflow impose any new unnecessary cognitive overhead to the team?

## Centralized Workflow

The Centralized Workflow is a great Git workflow for teams transitioning from SVN. Like Subversion, the Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project. Instead of trunk, the default development branch is called main and all changes are committed into this branch. This workflow doesn’t require any other branches besides main.

Transitioning to a distributed version control system may seem like a daunting task, but you don’t have to change your existing workflow to take advantage of Git. Your team can develop projects in the exact same way as they do with Subversion.

However, using Git to power your development workflow presents a few advantages over SVN. First, it gives every developer their own local copy of the entire project. This isolated environment lets each developer work independently of all other changes to a project - they can add commits to their local repository and completely forget about upstream developments until it's convenient for them.

Second, it gives you access to Git’s robust branching and merging model. Unlike SVN, Git branches are designed to be a fail-safe mechanism for integrating code and sharing changes between repositories. The Centralized Workflow is similar to other workflows in its utilization of a remote server-side hosted repository that developers push and pull form. Compared to other workflows, the Centralized Workflow has no defined pull request or forking patterns. A Centralized Workflow is generally better suited for teams migrating from SVN to Git and smaller size teams.

## How it works

Developers start by cloning the central repository. In their own local copies of the project, they edit files and commit changes as they would with SVN; however, these new commits are stored locally - they’re completely isolated from the central repository. This lets developers defer synchronizing upstream until they’re at a convenient break point.

To publish changes to the official project, developers "push" their local main branch to the central repository. This is the equivalent of svn commit, except that it adds all of the local commits that aren’t already in the central main branch.

### Initialize the central repository

Git Workflow: Initialize Central Bare Repository

First, someone needs to create the central repository on a server. If it’s a new project, you can initialize an empty repository. Otherwise, you’ll need to import an existing Git or SVN repository.

Central repositories should always be bare repositories (they shouldn’t have a working directory), which can be created as follows:

ssh user@host git init --bare /path/to/repo.git

Be sure to use a valid SSH username for user, the domain or IP address of your server for host, and the location where you'd like to store your repo for /path/to/repo.git. Note that the .git extension is conventionally appended to the repository name to indicate that it’s a bare repository.

### Hosted central repositories

Central repositories are often created through 3rd party Git hosting services like [Bitbucket Cloud](https://bitbucket.org/product) or [Bitbucket Server](https://bitbucket.org/product/enterprise). The process of initializing a bare repository discussed above is handled for you by the hosting service. The hosting service will then provide an address for the central repository to access from your local repository.

### Clone the central repository

Next, each developer creates a local copy of the entire project. This is accomplished via the git clone command:

git clone ssh://user@host/path/to/repo.git

When you clone a repository, Git automatically adds a shortcut called origin that points back to the “parent” repository, under the assumption that you'll want to interact with it further on down the road.

### Make changes and commit

Once the repository is cloned locally, a developer can make changes using the standard Git commit process: edit, stage, and commit. If you’re not familiar with the staging area, it’s a way to prepare a commit without having to include every change in the working directory. This lets you create highly focused commits, even if you’ve made a lot of local changes.

git status # View the state of the repo

git add <some-file> # Stage a file

git commit # Commit a file</some-file>

Remember that since these commands create local commits, John can repeat this process as many times as he wants without worrying about what’s going on in the central repository. This can be very useful for large features that need to be broken down into simpler, more atomic chunks.

### Push new commits to central repository

Once the local repository has new changes committed. These change will need to be pushed to share with other developers on the project.

git push origin main

This command will push the new committed changes to the central repository. When pushing changes to the central repository, it is possible that updates from another developer have been previously pushed that contain code which conflict with the intended push updates. Git will output a message indicating this conflict. In this situation, git pull will first need to be executed. This conflict scenario will be expanded on in the following section.

### Managing conflicts

The central repository represents the official project, so its commit history should be treated as sacred and immutable. If a developer’s local commits diverge from the central repository, Git will refuse to push their changes because this would overwrite official commits.

Git Workflows: Managing conflicts

Before the developer can publish their feature, they need to fetch the updated central commits and rebase their changes on top of them. This is like saying, “I want to add my changes to what everyone else has already done.” The result is a perfectly linear history, just like in traditional SVN workflows.

If local changes directly conflict with upstream commits, Git will pause the rebasing process and give you a chance to manually resolve the conflicts. The nice thing about Git is that it uses the same git status and git add commands for both generating commits and resolving merge conflicts. This makes it easy for new developers to manage their own merges. Plus, if they get themselves into trouble, Git makes it very easy to abort the entire rebase and try again (or go find help).

## Other common workflows

The Centralized Workflow is essentially a building block for other Git workflows. Most popular Git workflows will have some sort of centralized repo that individual developers will push and pull from. Below we will briefly discuss some other popular Git workflows. These extended workflows offer more specialized patterns in regard to managing branches for feature development, hot fixes, and eventual release.

## Feature branching

Feature Branching is a logical extension of Centralized Workflow. The core idea behind the [Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow) is that all feature development should take place in a dedicated branch instead of the main branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the main branch should never contain broken code, which is a huge advantage for continuous integration environments.

## Gitflow Workflow

The [Gitflow Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow) was first published in a highly regarded 2010 blog post from [Vincent Driessen at nvie](http://nvie.com/posts/a-successful-git-branching-model/). The Gitflow Workflow defines a strict branching model designed around the project release. This workflow doesn’t add any new concepts or commands beyond what’s required for the Feature Branch Workflow. Instead, it assigns very specific roles to different branches and defines how and when they should interact.

## Forking Workflow

The [Forking Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow) is fundamentally different than the other workflows discussed in this tutorial. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer a server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one.

## Guidelines

There is no one size fits all Git workflow. As previously stated, it’s important to develop a Git workflow that is a productivity enhancement for your team. In addition to team culture, a workflow should also complement business culture. Git features like branches and tags should complement your business’s release schedule. If your team is using [task tracking project management software](https://www.atlassian.com/software/jira) you may want to use branches that correspond with tasks in progress. In addition, some guidelines to consider when deciding on a workflow are:

### Short-lived branches

The longer a branch lives separate from the production branch, the higher the risk for merge conflicts and deployment challenges. Short-lived branches promote cleaner merges and deploys.

### Minimize and simplify reverts

It’s important to have a workflow that helps proactively prevent merges that will have to be reverted. A workflow that tests a branch before allowing it to be merged into the main branch is an example. However, accidents do happen. That being said, it’s beneficial to have a workflow that allows for easy reverts that will not disrupt the flow for other team members.

### Match a release schedule

A workflow should complement your business’s software development release cycle. If you plan to release multiple times a day, you will want to keep your main branch stable. If your release schedule is less frequent, you may want to consider using Git tags to tag a branch to a version.

* Contributing to a Project

### **Commit Guidelines**

Before we start looking at the specific use cases, here’s a quick note about commit messages. Having a good guideline for creating commits and sticking to it makes working with Git and collaborating with others a lot easier. The Git project provides a document that lays out a number of good tips for creating commits from which to submit patches — you can read it in the Git source code in the Documentation/SubmittingPatches file.

First, your submissions should not contain any whitespace errors. Git provides an easy way to check for this — before you commit, run git diff --check, which identifies possible whitespace errors and lists them for you.



Figure 56. Output of git diff --check

If you run that command before committing, you can tell if you’re about to commit whitespace issues that may annoy other developers.

Next, try to make each commit a logically separate changeset. If you can, try to make your changes digestible — don’t code for a whole weekend on five different issues and then submit them all as one massive commit on Monday. Even if you don’t commit during the weekend, use the staging area on Monday to split your work into at least one commit per issue, with a useful message per commit. If some of the changes modify the same file, try to use git add --patch to partially stage files (covered in detail in [Interactive Staging](https://git-scm.com/book/en/v2/ch00/_interactive_staging)). The project snapshot at the tip of the branch is identical whether you do one commit or five, as long as all the changes are added at some point, so try to make things easier on your fellow developers when they have to review your changes.

This approach also makes it easier to pull out or revert one of the changesets if you need to later. [Rewriting History](https://git-scm.com/book/en/v2/ch00/_rewriting_history) describes a number of useful Git tricks for rewriting history and interactively staging files — use these tools to help craft a clean and understandable history before sending the work to someone else.

The last thing to keep in mind is the commit message. Getting in the habit of creating quality commit messages makes using and collaborating with Git a lot easier. As a general rule, your messages should start with a single line that’s no more than about 50 characters and that describes the changeset concisely, followed by a blank line, followed by a more detailed explanation. The Git project requires that the more detailed explanation include your motivation for the change and contrast its implementation with previous behavior — this is a good guideline to follow. Write your commit message in the imperative: "Fix bug" and not "Fixed bug" or "Fixes bug." Here is a template you can follow, which we’ve lightly adapted from one [originally written by Tim Pope](https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html):

Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary. Wrap it to about 72

characters or so. In some contexts, the first line is treated as the

subject of an email and the rest of the text as the body. The blank

line separating the summary from the body is critical (unless you omit

the body entirely); tools like rebase will confuse you if you run the

two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug"

or "Fixes bug." This convention matches up with commit messages generated

by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, followed by a

single space, with blank lines in between, but conventions vary here

- Use a hanging indent

If all your commit messages follow this model, things will be much easier for you and the developers with whom you collaborate. 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.

| **Note** | **Do as we say, not as we do.**  For the sake of brevity, many of the examples in this book don’t have nicely-formatted commit messages like this; instead, we simply use the -m option to git commit.  In short, do as we say, not as we do. |
| --- | --- |

### **Private Small Team**

The simplest setup you’re likely to encounter is a private project with one or two other developers. “Private,” in this context, means closed-source — not accessible to the outside world. You and the other developers all have push access to the repository.

In this environment, you can follow a workflow similar to what you might do when using Subversion or another centralized system. You still get the advantages of things like offline committing and vastly simpler branching and merging, but the workflow can be very similar; the main difference is that merges happen client-side rather than on the server at commit time. Let’s see what it might look like when two developers start to work together with a shared repository. The first developer, John, clones the repository, makes a change, and commits locally. The protocol messages have been replaced with …​ in these examples to shorten them somewhat.

# John's Machine

$ git clone john@githost:simplegit.git

Cloning into 'simplegit'...

...

$ cd simplegit/

$ vim lib/simplegit.rb

$ git commit -am 'Remove invalid default value'

[master 738ee87] Remove invalid default value

1 files changed, 1 insertions(+), 1 deletions(-)

The second developer, Jessica, does the same thing — clones the repository and commits a change:

# Jessica's Machine

$ git clone jessica@githost:simplegit.git

Cloning into 'simplegit'...

...

$ cd simplegit/

$ vim TODO

$ git commit -am 'Add reset task'

[master fbff5bc] Add reset task

1 files changed, 1 insertions(+), 0 deletions(-)

Now, Jessica pushes her work to the server, which works just fine:

# Jessica's Machine

$ git push origin master

...

To jessica@githost:simplegit.git

1edee6b..fbff5bc master -> master

The last line of the output above shows a useful return message from the push operation. The basic format is <oldref>..<newref> fromref → toref, where oldref means the old reference, newref means the new reference, fromref is the name of the local reference being pushed, and toref is the name of the remote reference being updated. You’ll see similar output like this below in the discussions, so having a basic idea of the meaning will help in understanding the various states of the repositories. More details are available in the documentation for [git-push](https://git-scm.com/docs/git-push).

Continuing with this example, shortly afterwards, John makes some changes, commits them to his local repository, and tries to push them to the same server:

# John's Machine

$ git push origin master

To john@githost:simplegit.git

! [rejected] master -> master (non-fast forward)

error: failed to push some refs to 'john@githost:simplegit.git'

In this case, John’s push fails because of Jessica’s earlier push of **her** changes. 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.

As a first step, John fetches Jessica’s work (this only **fetches** Jessica’s upstream work, it does not yet merge it into John’s work):

$ git fetch origin

...

From john@githost:simplegit

+ 049d078...fbff5bc master -> origin/master

At this point, John’s local repository looks something like this:



Figure 57. John’s divergent history

Now John can merge Jessica’s work that he fetched into his own local work:

$ git merge origin/master

Merge made by the 'recursive' strategy.

TODO | 1 +

1 files changed, 1 insertions(+), 0 deletions(-)

As long as that local merge goes smoothly, John’s updated history will now look like this:



Figure 58. John’s repository after merging origin/master

At this point, John might want to test this new code to make sure none of Jessica’s work affects any of his and, as long as everything seems fine, he can finally push the new merged work up to the server:

$ git push origin master

...

To john@githost:simplegit.git

fbff5bc..72bbc59 master -> master

In the end, John’s commit history will look like this:



Figure 59. John’s history after pushing to the origin server

In the meantime, Jessica has created a new topic branch called issue54, and made three commits to that branch. She hasn’t fetched John’s changes yet, so her commit history looks like this:



Figure 60. Jessica’s topic branch

Suddenly, Jessica learns that John has pushed some new work to the server and she wants to take a look at it, so she can fetch all new content from the server that she does not yet have with:

# Jessica's Machine

$ git fetch origin

...

From jessica@githost:simplegit

fbff5bc..72bbc59 master -> origin/master

That pulls down the work John has pushed up in the meantime. Jessica’s history now looks like this:



Figure 61. Jessica’s history after fetching John’s changes

Jessica thinks her topic branch is ready, but she wants to know what part of John’s fetched work she has to merge into her work so that she can push. She runs git log to find out:

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

commit 738ee872852dfaa9d6634e0dea7a324040193016

Author: John Smith <jsmith@example.com>

Date: Fri May 29 16:01:27 2009 -0700

Remove invalid default value

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](https://git-scm.com/book/en/v2/ch00/_commit_ranges).

From the above output, we can see that there is a single commit that John has made that Jessica has not merged into her local work. If she merges origin/master, that is the single commit that will modify her local work.

Now, Jessica can merge her topic work into her master branch, merge John’s work (origin/master) into her master branch, and then push back to the server again.

First (having committed all of the work on her issue54 topic branch), Jessica switches back to her master branch in preparation for integrating all this work:

$ git checkout master

Switched to branch 'master'

Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.

Jessica can merge either origin/master or issue54 first — they’re both upstream, so the order doesn’t matter. The end snapshot should be identical no matter which order she chooses; only the history will be different. She chooses to merge the issue54 branch first:

$ git merge issue54

Updating fbff5bc..4af4298

Fast forward

README | 1 +

lib/simplegit.rb | 6 +++++-

2 files changed, 6 insertions(+), 1 deletions(-)

No problems occur; as you can see it was a simple fast-forward merge. Jessica now completes the local merging process by merging John’s earlier fetched work that is sitting in the origin/master branch:

$ git merge origin/master

Auto-merging lib/simplegit.rb

Merge made by the 'recursive' strategy.

lib/simplegit.rb | 2 +-

1 files changed, 1 insertions(+), 1 deletions(-)

Everything merges cleanly, and Jessica’s history now looks like this:



Figure 62. Jessica’s history after merging John’s changes

Now origin/master is reachable from Jessica’s master branch, so she should be able to successfully push (assuming John hasn’t pushed even more changes in the meantime):

$ git push origin master

...

To jessica@githost:simplegit.git

72bbc59..8059c15 master -> master

Each developer has committed a few times and merged each other’s work successfully.



Figure 63. Jessica’s history after pushing all changes back to the server

That is one of the simplest workflows. You work for a while (generally in a topic branch), and merge that work into your master branch when it’s ready to be integrated. When you want to share that work, you fetch and merge your master from origin/master if it has changed, and finally push to the master branch on the server. The general sequence is something like this:



Figure 64. General sequence of events for a simple multiple-developer Git workflow

### **Private Managed Team**

In this next scenario, you’ll look at contributor roles in a larger private group. You’ll learn how to work in an environment where small groups collaborate on features, after which those team-based contributions are integrated by another party.

Let’s say that John and Jessica are working together on one feature (call this “featureA”), while Jessica and a third developer, Josie, are working on a second (say, “featureB”). In this case, the company is using a type of integration-manager workflow where the work of the individual groups is integrated only by certain engineers, and the master branch of the main repo can be updated only by those engineers. In this scenario, all work is done in team-based branches and pulled together by the integrators later.

Let’s follow Jessica’s workflow as she works on her two features, collaborating in parallel with two different developers in this environment. Assuming she already has her repository cloned, she decides to work on featureA first. She creates a new branch for the feature and does some work on it there:

# Jessica's Machine

$ git checkout -b featureA

Switched to a new branch 'featureA'

$ vim lib/simplegit.rb

$ git commit -am 'Add limit to log function'

[featureA 3300904] Add limit to log function

1 files changed, 1 insertions(+), 1 deletions(-)

At this point, she needs to share her work with John, so she pushes her featureA branch commits up to the server. Jessica doesn’t have push access to the master branch — only the integrators do — so she has to push to another branch in order to collaborate with John:

$ git push -u origin featureA

...

To jessica@githost:simplegit.git

\* [new branch] featureA -> featureA

Jessica emails John to tell him that she’s pushed some work into a branch named featureA and he can look at it now. While she waits for feedback from John, Jessica decides to start working on featureB with Josie. To begin, she starts a new feature branch, basing it off the server’s master branch:

# Jessica's Machine

$ git fetch origin

$ git checkout -b featureB origin/master

Switched to a new branch 'featureB'

Now, Jessica makes a couple of commits on the featureB branch:

$ vim lib/simplegit.rb

$ git commit -am 'Make ls-tree function recursive'

[featureB e5b0fdc] Make ls-tree function recursive

1 files changed, 1 insertions(+), 1 deletions(-)

$ vim lib/simplegit.rb

$ git commit -am 'Add ls-files'

[featureB 8512791] Add ls-files

1 files changed, 5 insertions(+), 0 deletions(-)

Jessica’s repository now looks like this:



Figure 65. Jessica’s initial commit history

She’s ready to push her work, but gets an email from Josie that a branch with some initial “featureB” work on it was already pushed to the server as the featureBee branch. Jessica needs to merge those changes with her own before she can push her work to the server. Jessica first fetches Josie’s changes with git fetch:

$ git fetch origin

...

From jessica@githost:simplegit

\* [new branch] featureBee -> origin/featureBee

Assuming Jessica is still on her checked-out featureB branch, she can now merge Josie’s work into that branch with git merge:

$ git merge origin/featureBee

Auto-merging lib/simplegit.rb

Merge made by the 'recursive' strategy.

lib/simplegit.rb | 4 ++++

1 files changed, 4 insertions(+), 0 deletions(-)

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:featureBee

...

To jessica@githost:simplegit.git

fba9af8..cd685d1 featureB -> featureBee

This is called a **refspec**. See [The Refspec](https://git-scm.com/book/en/v2/ch00/_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, which configures the branches for easier pushing and pulling later.

Suddenly, Jessica gets email from John, who tells her he’s pushed some changes to the featureA branch on which they are collaborating, and he asks Jessica to take a look at them. Again, Jessica runs a simple git fetch to fetch **all** new content from the server, including (of course) John’s latest work:

$ git fetch origin

...

From jessica@githost:simplegit

3300904..aad881d featureA -> origin/featureA

Jessica can display the log of John’s new work by comparing the content of the newly-fetched featureA branch with her local copy of the same branch:

$ git log featureA..origin/featureA

commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6

Author: John Smith <jsmith@example.com>

Date: Fri May 29 19:57:33 2009 -0700

Increase log output to 30 from 25

If Jessica likes what she sees, she can merge John’s new work into her local featureA branch with:

$ git checkout featureA

Switched to branch 'featureA'

$ git merge origin/featureA

Updating 3300904..aad881d

Fast forward

lib/simplegit.rb | 10 +++++++++-

1 files changed, 9 insertions(+), 1 deletions(-)

Finally, Jessica might want to make a couple minor changes to all that merged content, so she is free to make those changes, commit them to her local featureA branch, and push the end result back to the server:

$ git commit -am 'Add small tweak to merged content'

[featureA 774b3ed] Add small tweak to merged content

1 files changed, 1 insertions(+), 1 deletions(-)

$ git push

...

To jessica@githost:simplegit.git

3300904..774b3ed featureA -> featureA

Jessica’s commit history now looks something like this:



Figure 66. Jessica’s history after committing on a feature branch

At some point, Jessica, Josie, and John inform the integrators that the featureA and featureBee branches on the server are ready for integration into the mainline. After the integrators merge these branches into the mainline, a fetch will bring down the new merge commit, making the history look like this:



Figure 67. Jessica’s history after merging both her topic branches

Many groups switch to Git because of this ability to have multiple teams working in parallel, merging the different lines of work late in the process. The ability of smaller subgroups of a team to collaborate via remote branches without necessarily having to involve or impede the entire team is a huge benefit of Git. The sequence for the workflow you saw here is something like this:



Figure 68. Basic sequence of this managed-team workflow

### **Forked Public Project**

Contributing to public projects is a bit different. Because you don’t have the permissions to directly update branches on the project, you have to get the work to the maintainers some other way. This first example describes contributing via forking on Git hosts that support easy forking. Many hosting sites support this (including GitHub, BitBucket, repo.or.cz, and others), and many project maintainers expect this style of contribution. The next section deals with projects that prefer to accept contributed patches via email.

First, you’ll probably want to clone the main repository, create a topic branch for the patch or patch series you’re planning to contribute, and do your work there. The sequence looks basically like this:

$ git clone <url>

$ cd project

$ git checkout -b featureA

... work ...

$ git commit

... work ...

$ git commit

| **Note** | You may want to use rebase -i to squash your work down to a single commit, or rearrange the work in the commits to make the patch easier for the maintainer to review — see [Rewriting History](https://git-scm.com/book/en/v2/ch00/_rewriting_history) for more information about interactive rebasing. |
| --- | --- |

When your branch work is finished and you’re ready to contribute it back to the maintainers, go to the original project page and click the “Fork” button, creating your own writable fork of the project. You then need to add this repository URL as a new remote of your local repository; in this example, let’s call it myfork:

$ git remote add myfork <url>

You then need to push your new work to this repository. It’s easiest to push the topic branch you’re working on to your forked repository, rather than merging that work into your master branch and pushing that. The reason is that if your work isn’t accepted or is cherry-picked, you don’t have to rewind your master branch (the Git cherry-pick operation is covered in more detail in [Rebasing and Cherry-Picking Workflows](https://git-scm.com/book/en/v2/ch00/_rebase_cherry_pick)). If the maintainers merge, rebase, or cherry-pick your work, you’ll eventually get it back via pulling from their repository anyhow.

In any event, you can push your work with:

$ git push -u myfork featureA

Once your work has been pushed to your fork of the repository, you need to notify the maintainers of the original project that you have work you’d like them to merge. This is often called a **pull request**, and you typically generate such a request either via the website — GitHub has its own “Pull Request” mechanism that we’ll go over in [GitHub](https://git-scm.com/book/en/v2/ch00/ch06-github) — or you can run the git request-pull command and email the subsequent output to the project maintainer manually.

The git request-pull command takes the base branch into which you want your topic branch pulled and the Git repository URL you want them to pull from, and produces a summary of all the changes you’re asking to be pulled. For instance, if Jessica wants to send John a pull request, and she’s done two commits on the topic branch she just pushed, she can run this:

$ git request-pull origin/master myfork

The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:

Jessica Smith (1):

Create new function

are available in the git repository at:

git://githost/simplegit.git featureA

Jessica Smith (2):

Add limit to log function

Increase log output to 30 from 25

lib/simplegit.rb | 10 +++++++++-

1 files changed, 9 insertions(+), 1 deletions(-)

This output can be sent to the maintainer — it tells them where the work was branched from, summarizes the commits, and identifies from where the new work is to be pulled.

On a project for which you’re not the maintainer, it’s generally easier to have a branch like master always track origin/master and to do your work in topic branches that you can easily discard if they’re rejected. Having work themes isolated into topic branches also makes it easier for you to rebase your work if the tip of the main repository has moved in the meantime and your commits no longer apply cleanly. For example, if you want to submit a second topic of work to the project, don’t continue working on the topic branch you just pushed up — start over from the main repository’s master branch:

$ git checkout -b featureB origin/master

... work ...

$ git commit

$ git push myfork featureB

$ git request-pull origin/master myfork

... email generated request pull to maintainer ...

$ git fetch origin

Now, each of your topics is contained within a silo — similar to a patch queue — that you can rewrite, rebase, and modify without the topics interfering or interdepending on each other, like so:



Figure 69. Initial commit history with featureB work

Let’s say the project maintainer has pulled in a bunch of other patches and tried your first branch, but it no longer cleanly merges. In this case, you can try to rebase that branch on top of origin/master, resolve the conflicts for the maintainer, and then resubmit your changes:

$ git checkout featureA

$ git rebase origin/master

$ git push -f myfork featureA

This rewrites your history to now look like [Commit history after featureA work](https://git-scm.com/book/en/v2/ch00/psp_b).



Figure 70. Commit history after featureA work

Because you rebased the branch, you have to specify the -f to your push command in order to be able to replace the featureA branch on the server with a commit that isn’t a descendant of it. An alternative would be to push this new work to a different branch on the server (perhaps called featureAv2).

Let’s look at one more possible scenario: the maintainer has looked at work in your second branch and likes the concept but would like you to change an implementation detail. You’ll also take this opportunity to move the work to be based off the project’s current master branch. 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.



Figure 71. Commit history after featureBv2 work

### **Public Project over Email**

Many projects have established procedures for accepting patches — you’ll need to check the specific rules for each project, because they will differ. Since there are several older, larger projects which accept patches via a developer mailing list, we’ll go over an example of that now.

The workflow is similar to the previous use case — you create topic branches for each patch series you work on. The difference is how you submit them to the project. Instead of forking the project and pushing to your own writable version, you generate email versions of each commit series and email them to the developer mailing list:

$ git checkout -b topicA

... work ...

$ git commit

... work ...

$ git commit

Now you have two commits that you want to send to the mailing list. You use git format-patch to generate the mbox-formatted files that you can email to the list — it turns each commit into an email message with the first line of the commit message as the subject and the rest of the message plus the patch that the commit introduces as the body. The nice thing about this is that applying a patch from an email generated with format-patch preserves all the commit information properly.

$ git format-patch -M origin/master

0001-add-limit-to-log-function.patch

0002-increase-log-output-to-30-from-25.patch

The format-patch command prints out the names of the patch files it creates. The -M switch tells Git to look for renames. The files end up looking like this:

$ cat 0001-add-limit-to-log-function.patch

From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001

From: Jessica Smith <jessica@example.com>

Date: Sun, 6 Apr 2008 10:17:23 -0700

Subject: [PATCH 1/2] Add limit to log function

Limit log functionality to the first 20

---

lib/simplegit.rb | 2 +-

1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index 76f47bc..f9815f1 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -14,7 +14,7 @@ class SimpleGit

end

def log(treeish = 'master')

- command("git log #{treeish}")

+ command("git log -n 20 #{treeish}")

end

def ls\_tree(treeish = 'master')

--

2.1.0

You can also edit these patch files to add more information for the email list that you don’t want to show up in the commit message. If you add text between the --- line and the beginning of the patch (the diff --git line), the developers can read it, but that content is ignored by the patching process.

To email this to a mailing list, you can either paste the file into your email program or send it via a command-line program. Pasting the text often causes formatting issues, especially with “smarter” clients that don’t preserve newlines and other whitespace appropriately. Luckily, Git provides a tool to help you send properly formatted patches via IMAP, which may be easier for you. We’ll demonstrate how to send a patch via Gmail, which happens to be the email agent we know best; you can read detailed instructions for a number of mail programs at the end of the aforementioned Documentation/SubmittingPatches file in the Git source code.

First, you need to set up the imap section in your ~/.gitconfig file. You can set each value separately with a series of git config commands, or you can add them manually, but in the end your config file should look something like this:

[imap]

folder = "[Gmail]/Drafts"

host = imaps://imap.gmail.com

user = user@gmail.com

pass = YX]8g76G\_2^sFbd

port = 993

sslverify = false

If your IMAP server doesn’t use SSL, the last two lines probably aren’t necessary, and the host value will be imap:// instead of imaps://. When that is set up, you can use git imap-send to place the patch series in the Drafts folder of the specified IMAP server:

$ cat \*.patch |git imap-send

Resolving imap.gmail.com... ok

Connecting to [74.125.142.109]:993... ok

Logging in...

sending 2 messages

100% (2/2) done

At this point, you should be able to go to your Drafts folder, change the To field to the mailing list you’re sending the patch to, possibly CC the maintainer or person responsible for that section, and send it off.

You can also send the patches through an SMTP server. As before, you can set each value separately with a series of git config commands, or you can add them manually in the sendemail section in your ~/.gitconfig file:

[sendemail]

smtpencryption = tls

smtpserver = smtp.gmail.com

smtpuser = user@gmail.com

smtpserverport = 587

After this is done, you can use git send-email to send your patches:

$ git send-email \*.patch

0001-add-limit-to-log-function.patch

0002-increase-log-output-to-30-from-25.patch

Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]

Emails will be sent from: Jessica Smith <jessica@example.com>

Who should the emails be sent to? jessica@example.com

Message-ID to be used as In-Reply-To for the first email? y

Then, Git spits out a bunch of log information looking something like this for each patch you’re sending:

(mbox) Adding cc: Jessica Smith <jessica@example.com> from

\line 'From: Jessica Smith <jessica@example.com>'

OK. Log says:

Sendmail: /usr/sbin/sendmail -i jessica@example.com

From: Jessica Smith <jessica@example.com>

To: jessica@example.com

Subject: [PATCH 1/2] Add limit to log function

Date: Sat, 30 May 2009 13:29:15 -0700

Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>

X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty

In-Reply-To: <y>

References: <y>

Result: OK

* Maintaining a Project

### **Working in Topic Branches**

When you’re thinking of integrating new work, it’s generally a good idea to try it out in a **topic branch** — a temporary branch specifically made to try out that new work. This way, it’s easy to tweak a patch individually and leave it if it’s not working until you have time to come back to it. If you create a simple branch name based on the theme of the work you’re going to try, such as ruby\_client or something similarly descriptive, you can easily remember it if you have to abandon it for a while and come back later. The maintainer of the Git project tends to namespace these branches as well — such as sc/ruby\_client, where sc is short for the person who contributed the work. As you’ll remember, you can create the branch based off your master branch like this:

$ git branch sc/ruby\_client master

Or, if you want to also switch to it immediately, you can use the checkout -b option:

$ git checkout -b sc/ruby\_client master

Now you’re ready to add the contributed work that you received into this topic branch and determine if you want to merge it into your longer-term branches.

### **Applying Patches from Email**

If you receive a patch over email that you need to integrate into your project, you need to apply the patch in your topic branch to evaluate it. There are two ways to apply an emailed patch: with git apply or with git am.

#### **Applying a Patch with apply**

If you received the patch from someone who generated it with git diff or some variation of the Unix diff command (which is not recommended; see the next section), you can apply it with the git apply command. Assuming you saved the patch at /tmp/patch-ruby-client.patch, you can apply the patch like this:

$ git apply /tmp/patch-ruby-client.patch

This modifies the files in your working directory. It’s almost identical to running a patch -p1 command to apply the patch, although it’s more paranoid and accepts fewer fuzzy matches than patch. It also handles file adds, deletes, and renames if they’re described in the git diff format, which patch won’t do. Finally, git apply is an “apply all or abort all” model where either everything is applied or nothing is, whereas patch can partially apply patchfiles, leaving your working directory in a weird state. git apply is overall much more conservative than patch. It won’t create a commit for you — after running it, you must stage and commit the changes introduced manually.

You can also use git apply to see if a patch applies cleanly before you try actually applying it — you can run git apply --check with the patch:

$ git apply --check 0001-see-if-this-helps-the-gem.patch

error: patch failed: ticgit.gemspec:1

error: ticgit.gemspec: patch does not apply

If there is no output, then the patch should apply cleanly. This command also exits with a non-zero status if the check fails, so you can use it in scripts if you wan

### **Git Tools**

Revision Selection

### **Single Revisions**

You can obviously refer to any single commit by its full, 40-character SHA-1 hash, but there are more human-friendly ways to refer to commits as well. This section outlines the various ways you can refer to any commit.

### **Short SHA-1**

Git is smart enough to figure out what commit you’re referring to if you provide the first few characters of the SHA-1 hash, as long as that partial hash is at least four characters long and unambiguous; that is, no other object in the object database can have a hash that begins with the same prefix.

For example, to examine a specific commit where you know you added certain functionality, you might first run the git log command to locate the commit:

$ git log

commit 734713bc047d87bf7eac9674765ae793478c50d3

Author: Scott Chacon <schacon@gmail.com>

Date: Fri Jan 2 18:32:33 2009 -0800

Fix refs handling, add gc auto, update tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef

Merge: 1c002dd... 35cfb2b...

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Dec 11 14:58:32 2008 -0800

Add some blame and merge stuff

In this case, say you’re interested in the commit whose hash begins with 1c002dd…​. You can inspect that commit with any of the following variations of git show (assuming the shorter versions are unambiguous):

$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b

$ git show 1c002dd4b536e7479f

$ git show 1c002d

**Interactive Staging**

### **Staging and Unstaging Files**

**git add -i**

If you type u or 2 (for update) at the What now> prompt, you’re prompted for which files you want to stage:

What now> u

staged unstaged path

1: unchanged +0/-1 TODO

2: unchanged +1/-1 index.html

3: unchanged +5/-1 lib/simplegit.rb

Update>>

To stage the TODO and index.html files, you can type the numbers:

Update>> 1,2

staged unstaged path

\* 1: unchanged +0/-1 TODO

\* 2: unchanged +1/-1 index.html

3: unchanged +5/-1 lib/simplegit.rb

Update>>

The \* next to each file means the file is selected to be staged. If you press Enter after typing nothing at the Update>> prompt, Git takes anything selected and stages it for you:

Update>>

updated 2 paths

\*\*\* Commands \*\*\*

1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd untracked

5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp

What now> s

staged unstaged path

1: +0/-1 nothing TODO

2: +1/-1 nothing index.html

3: unchanged +5/-1 lib/simplegit.rb

Now you can see that the TODO and index.html files are staged and the simplegit.rb file is still unstaged. If you want to unstage the TODO file at this point, you use the r or 3 (for revert) option:

\*\*\* Commands \*\*\*

1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd untracked

5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp

What now> r

staged unstaged path

1: +0/-1 nothing TODO

2: +1/-1 nothing index.html

3: unchanged +5/-1 lib/simplegit.rb

Revert>> 1

staged unstaged path

\* 1: +0/-1 nothing TODO

2: +1/-1 nothing index.html

3: unchanged +5/-1 lib/simplegit.rb

Revert>> [enter]

reverted one path

Looking at your Git status again, you can see that you’ve unstaged the TODO file:

\*\*\* Commands \*\*\*

1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd untracked

5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp

What now> s

staged unstaged path

1: unchanged +0/-1 TODO

2: +1/-1 nothing index.html

3: unchanged +5/-1 lib/simplegit.rb

To see the diff of what you’ve staged, you can use the d or 6 (for diff) command. It shows you a list of your staged files, and you can select the ones for which you would like to see the staged diff. This is much like specifying git diff --cached on the command line:

\*\*\* Commands \*\*\*

1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd untracked

5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp

What now> d

staged unstaged path

1: +1/-1 nothing index.html

Review diff>> 1

diff --git a/index.html b/index.html

index 4d07108..4335f49 100644

--- a/index.html

+++ b/index.html

@@ -16,7 +16,7 @@ Date Finder

<p id="out">...</p>

-<div id="footer">contact : support@github.com</div>

+<div id="footer">contact : email.support@github.com</div>

<script type="text/javascript">

With these basic commands, you can use the interactive add mode to deal with your staging area a little more easily.

### **Staging Patches**

It’s also possible for Git to stage certain **parts** of files and not the rest. For example, if you make two changes to your simplegit.rb file and want to stage one of them and not the other, doing so is very easy in Git. From the same interactive prompt explained in the previous section, type p or 5 (for patch). Git will ask you which files you would like to partially stage; then, for each section of the selected files, it will display hunks of the file diff and ask if you would like to stage them, one by one:

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index dd5ecc4..57399e0 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -22,7 +22,7 @@ class SimpleGit

end

def log(treeish = 'master')

- command("git log -n 25 #{treeish}")

+ command("git log -n 30 #{treeish}")

end

def blame(path)

Stage this hunk [y,n,a,d,/,j,J,g,e,?]?

You have a lot of options at this point. Typing ? shows a list of what you can do:

Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?

y - stage this hunk

n - do not stage this hunk

a - stage this and all the remaining hunks in the file

d - do not stage this hunk nor any of the remaining hunks in the file

g - select a hunk to go to

/ - search for a hunk matching the given regex

j - leave this hunk undecided, see next undecided hunk

J - leave this hunk undecided, see next hunk

k - leave this hunk undecided, see previous undecided hunk

K - leave this hunk undecided, see previous hunk

s - split the current hunk into smaller hunks

e - manually edit the current hunk

? - print help

Generally, you’ll type y or n if you want to stage each hunk, but staging all of them in certain files or skipping a hunk decision until later can be helpful too. If you stage one part of the file and leave another part unstaged, your status output will look like this:

What now> 1

staged unstaged path

1: unchanged +0/-1 TODO

2: +1/-1 nothing index.html

3: +1/-1 +4/-0 lib/simplegit.rb

The status of the simplegit.rb file is interesting. It shows you that a couple of lines are staged and a couple are unstaged. You’ve partially staged this file. At this point, you can exit the interactive adding script and run git commit to commit the partially staged files.

You also don’t need to be in interactive add mode to do the partial-file staging — you can start the same script by using git add -p or git add --patch on the command line.

**Rewriting History**

### **Changing the Last Commit**

Changing your most recent commit is probably the most common rewriting of history that you’ll do. You’ll often want to do two basic things to your last commit: simply change the commit message, or change the actual content of the commit by adding, removing and modifying files.

If you simply want to modify your last commit message, that’s easy:

$ **git commit --amend**

The command above loads the previous commit message into an editor session, where you can make changes to the message, save those changes and exit. When you save and close the editor, the editor writes a new commit containing that updated commit message and makes it your new last commit.

If, on the other hand, you want to change the actual **content** of your last commit, the process works basically the same way — first make the changes you think you forgot, stage those changes, and the subsequent git commit --amend **replaces** that last commit with your new, improved commit.

You need to be careful with this technique because amending changes the SHA-1 of the commit. It’s like a very small rebase — don’t amend your last commit if you’ve already pushed it.

| **Tip** | **An amended commit may (or may not) need an amended commit message**  When you amend a commit, you have the opportunity to change both the commit message and the content of the commit. If you amend the content of the commit substantially, you should almost certainly update the commit message to reflect that amended content.  On the other hand, if your amendments are suitably trivial (fixing a silly typo or adding a file you forgot to stage) such that the earlier commit message is just fine, you can simply make the changes, stage them, and avoid the unnecessary editor session entirely with:  $ git commit --amend --no-edit |
| --- | --- |

### 

### **Changing Multiple Commit Messages**

To modify a commit that is farther back in your history, you must move to more complex tools. Git doesn’t have a modify-history tool, but you can use the rebase tool to rebase a series of commits onto the HEAD that they were originally based on instead of moving them to another one. With the interactive rebase tool, you can then stop after each commit you want to modify and change the message, add files, or do whatever you wish. You can run rebase interactively by adding the -i option to git rebase. You must indicate how far back you want to rewrite commits by telling the command which commit to rebase onto.

For example, if you want to change the last three commit messages, or any of the commit messages in that group, you supply as an argument to git rebase -i the parent of the last commit you want to edit, which is HEAD~2^ or HEAD~3. It may be easier to remember the ~3 because you’re trying to edit the last three commits, but keep in mind that you’re actually designating four commits ago, the parent of the last commit you want to edit:

$ git rebase -i HEAD~3

Remember again that this is a rebasing command — every commit in the range HEAD~3..HEAD with a changed message **and all of its descendants** will be rewritten. Don’t include any commit you’ve already pushed to a central server — doing so will confuse other developers by providing an alternate version of the same change.

**Debugging with Git (bisect)**

Use binary search to find the commit that introduced a bug

Automatically bisect a broken build between v1.2 and HEAD:  
$ git bisect start HEAD v1.2 -- # HEAD is bad, v1.2 is good

$ git bisect run make # "make" builds the app

* $ git bisect reset # quit the bisect session

Automatically bisect a test failure between origin and HEAD:  
$ git bisect start HEAD origin -- # HEAD is bad, origin is good

$ git bisect run make test # "make test" builds and tests

* $ git bisect reset # quit the bisect session

Automatically bisect a broken test case:  
$ cat ~/test.sh

#!/bin/sh

make || exit 125 # this skips broken builds

~/check\_test\_case.sh # does the test case pass?

$ git bisect start HEAD HEAD~10 -- # culprit is among the last 10

$ git bisect run ~/test.sh

* $ git bisect reset # quit the bisect session  
  Here we use a test.sh custom script. In this script, if make fails, we skip the current commit. check\_test\_case.sh should exit 0 if the test case passes, and exit 1 otherwise.  
  It is safer if both test.sh and check\_test\_case.sh are outside the repository to prevent interactions between the bisect, make and test processes and the scripts.

Automatically bisect with temporary modifications (hot-fix):  
$ cat ~/test.sh

#!/bin/sh

# tweak the working tree by merging the hot-fix branch

# and then attempt a build

if git merge --no-commit --no-ff hot-fix &&

make

then

# run project specific test and report its status

~/check\_test\_case.sh

status=$?

else

# tell the caller this is untestable

status=125

fi

# undo the tweak to allow clean flipping to the next commit

git reset --hard

# return control

* exit $status  
  This applies modifications from a hot-fix branch before each test run, e.g. in case your build or test environment changed so that older revisions may need a fix which newer ones have already. (Make sure the hot-fix branch is based off a commit which is contained in all revisions which you are bisecting, so that the merge does not pull in too much, or use git cherry-pick instead of git merge.)

Automatically bisect a broken test case:  
$ git bisect start HEAD HEAD~10 -- # culprit is among the last 10

$ git bisect run sh -c "make || exit 125; ~/check\_test\_case.sh"

* $ git bisect reset # quit the bisect session  
  This shows that you can do without a run script if you write the test on a single line.

Locate a good region of the object graph in a damaged repository  
$ git bisect start HEAD <known-good-commit> [ <boundary-commit> ... ] --no-checkout

$ git bisect run sh -c '

GOOD=$(git for-each-ref "--format=%(objectname)" refs/bisect/good-\*) &&

git rev-list --objects BISECT\_HEAD --not $GOOD >tmp.$$ &&

git pack-objects --stdout >/dev/null <tmp.$$

rc=$?

rm -f tmp.$$

test $rc = 0'

* $ git bisect reset # quit the bisect session  
  In this case, when **git bisect run** finishes, bisect/bad will refer to a commit that has at least one parent whose reachable graph is fully traversable in the sense required by **git pack objects**.

Look for a fix instead of a regression in the code  
$ git bisect start

$ git bisect new HEAD # current commit is marked as new

* $ git bisect old HEAD~10 # the tenth commit from now is marked as old  
  or:

$ git bisect start --term-old broken --term-new fixed

$ git bisect fixed

$ git bisect broken HEAD~10

**Submodules**

It often happens that while working on one project, you need to use another project from within it. Perhaps it’s a library that a third party developed or that you’re developing separately and using in multiple parent projects. A common issue arises in these scenarios: you want to be able to treat the two projects as separate yet still be able to use one from within the other.

Here’s an example. Suppose you’re developing a website and creating Atom feeds. Instead of writing your own Atom-generating code, you decide to use a library. You’re likely to have to either include this code from a shared library like a CPAN install or Ruby gem, or copy the source code into your own project tree. The issue with including the library is that it’s difficult to customize the library in any way and often more difficult to deploy it, because you need to make sure every client has that library available. The issue with copying the code into your own project is that any custom changes you make are difficult to merge when upstream changes become available.

Git addresses this issue using submodules. Submodules allow you to keep a Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

### **Starting with Submodules**

We’ll walk through developing a simple project that has been split up into a main project and a few sub-projects.

Let’s start by adding an existing Git repository as a submodule of the repository that we’re working on. To add a new submodule you use the git submodule add command with the absolute or relative URL of the project you would like to start tracking. In this example, we’ll add a library called “DbConnector”.

**https://github.com/apache/apr-util.git**

**https://gitlab.com/tortoisegit/apr.git**

**https://gitlab.com/tortoisegit/tgit.git**

**https://github.com/madler/zlib.git**

**https://github.com/libgit2/libgit2.git**

**http://anongit.freedesktop.org/git/libreoffice/dictionaries.git**

**https://github.com/editorconfig/editorconfig-core-c.git**

**https://gitlab.com/tortoisegit/pcre.git**

**https://chromium.googlesource.com/external/gtest.git**

$ git submodule add https://github.com/apache/apr-util.git

Cloning into 'DbConnector'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.

By default, submodules will add the subproject into a directory named the same as the repository, in this case “DbConnector”. You can add a different path at the end of the command if you want it to go elsewhere.

If you run git status at this point, you’ll notice a few things.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: .gitmodules

new file: DbConnector

First you should notice the new .gitmodules file. This is a configuration file that stores the mapping between the project’s URL and the local subdirectory you’ve pulled it into:

[submodule "DbConnector"]

path = DbConnector

url = https://github.com/chaconinc/DbConnector

If you have multiple submodules, you’ll have multiple entries in this file. It’s important to note that this file is version-controlled with your other files, like your .gitignore file. It’s pushed and pulled with the rest of your project. This is how other people who clone this project know where to get the submodule projects from.

| **Note** | Since the URL in the .gitmodules file is what other people will first try to clone/fetch from, make sure to use a URL that they can access if possible. For example, if you use a different URL to push to than others would to pull from, use the one that others have access to. You can overwrite this value locally with git config submodule.DbConnector.url PRIVATE\_URL for your own use. When applicable, a relative URL can be helpful. |
| --- | --- |

The other listing in the git status output is the project folder entry. If you run git diff on that, you see something interesting:

$ git diff --cached DbConnector

diff --git a/DbConnector b/DbConnector

new file mode 160000

index 0000000..c3f01dc

--- /dev/null

+++ b/DbConnector

@@ -0,0 +1 @@

+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc

Although DbConnector is a subdirectory in your working directory, Git sees it as a submodule and doesn’t track its contents when you’re not in that directory. Instead, Git sees it as a particular commit from that repository.

If you want a little nicer diff output, you can pass the --submodule option to git diff.

$ git diff --cached --submodule

diff --git a/.gitmodules b/.gitmodules

new file mode 100644

index 0000000..71fc376

--- /dev/null

+++ b/.gitmodules

@@ -0,0 +1,3 @@

+[submodule "DbConnector"]

+ path = DbConnector

+ url = https://github.com/chaconinc/DbConnector

Submodule DbConnector 0000000...c3f01dc (new submodule)

When you commit, you see something like this:

$ git commit -am 'Add DbConnector module'

[master fb9093c] Add DbConnector module

2 files changed, 4 insertions(+)

create mode 100644 .gitmodules

create mode 160000 DbConnector

Notice the 160000 mode for the DbConnector entry. That is a special mode in Git that basically means you’re recording a commit as a directory entry rather than a subdirectory or a file.

Lastly, push these changes:

$ git push origin master

### **Cloning a Project with Submodules**

Here we’ll clone a project with a submodule in it. When you clone such a project, by default you get the directories that contain submodules, but none of the files within them yet:

$ git clone https://github.com/manaswinidas/Docs-test

Cloning into 'MainProject'...

remote: Counting objects: 14, done.

remote: Compressing objects: 100% (13/13), done.

remote: Total 14 (delta 1), reused 13 (delta 0)

Unpacking objects: 100% (14/14), done.

Checking connectivity... done.

$ cd MainProject

$ ls -la

total 16

drwxr-xr-x 9 schacon staff 306 Sep 17 15:21 .

drwxr-xr-x 7 schacon staff 238 Sep 17 15:21 ..

drwxr-xr-x 13 schacon staff 442 Sep 17 15:21 .git

-rw-r--r-- 1 schacon staff 92 Sep 17 15:21 .gitmodules

drwxr-xr-x 2 schacon staff 68 Sep 17 15:21 DbConnector

-rw-r--r-- 1 schacon staff 756 Sep 17 15:21 Makefile

drwxr-xr-x 3 schacon staff 102 Sep 17 15:21 includes

drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 scripts

drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 src

$ cd DbConnector/

$ ls

$

The DbConnector directory is there, but empty. You must run two commands: git submodule init to initialize your local configuration file, and git submodule update to fetch all the data from that project and check out the appropriate commit listed in your superproject:

$ git submodule init

Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'

$ git submodule update

Cloning into 'DbConnector'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.

Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'

Now your DbConnector subdirectory is at the exact state it was in when you committed earlier.

There is another way to do this which is a little simpler, however. If you pass --recurse-submodules to the git clone command, it will automatically initialize and update each submodule in the repository, including nested submodules if any of the submodules in the repository have submodules themselves.

$ git clone --recurse-submodules https://github.com/chaconinc/MainProject

Cloning into 'MainProject'...

remote: Counting objects: 14, done.

remote: Compressing objects: 100% (13/13), done.

remote: Total 14 (delta 1), reused 13 (delta 0)

Unpacking objects: 100% (14/14), done.

Checking connectivity... done.

Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'

Cloning into 'DbConnector'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.

Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'

If you already cloned the project and forgot --recurse-submodules, you can combine the git submodule init and git submodule update steps by running git submodule update --init. To also initialize, fetch and checkout any nested submodules, you can use the foolproof git submodule update --init --recursive.

### **Working on a Project with Submodules**

Now we have a copy of a project with submodules in it and will collaborate with our teammates on both the main project and the submodule project.

#### **Pulling in Upstream Changes from the Submodule Remote**

The simplest model of using submodules in a project would be if you were simply consuming a subproject and wanted to get updates from it from time to time but were not actually modifying anything in your checkout. Let’s walk through a simple example there.

If you want to check for new work in a submodule, you can go into the directory and run git fetch and git merge the upstream branch to update the local code.

$ git fetch

From https://github.com/chaconinc/DbConnector

c3f01dc..d0354fc master -> origin/master

$ git merge origin/master

Updating c3f01dc..d0354fc

Fast-forward

scripts/connect.sh | 1 +

src/db.c | 1 +

2 files changed, 2 insertions(+)

Now if you go back into the main project and run git diff --submodule you can see that the submodule was updated and get a list of commits that were added to it. If you don’t want to type --submodule every time you run git diff, you can set it as the default format by setting the diff.submodule config value to “log”.

$ git config --global diff.submodule log

$ git diff

Submodule DbConnector c3f01dc..d0354fc:

> more efficient db routine

> better connection routine

If you commit at this point then you will lock the submodule into having the new code when other people update.

There is an easier way to do this as well, if you prefer to not manually fetch and merge in the subdirectory. If you run git submodule update --remote, Git will go into your submodules and fetch and update for you.

$ git submodule update --remote DbConnector

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 4 (delta 2), reused 4 (delta 2)

Unpacking objects: 100% (4/4), done.

From https://github.com/chaconinc/DbConnector

3f19983..d0354fc master -> origin/master

Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c0644'

This command will by default assume that you want to update the checkout to the master branch of the submodule repository. You can, however, set this to something different if you want. For example, if you want to have the DbConnector submodule track that repository’s “stable” branch, you can set it in either your .gitmodules file (so everyone else also tracks it), or just in your local .git/config file. Let’s set it in the .gitmodules file:

$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 4 (delta 2), reused 4 (delta 2)

Unpacking objects: 100% (4/4), done.

From https://github.com/chaconinc/DbConnector

27cf5d3..c87d55d stable -> origin/stable

Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'

If you leave off the -f .gitmodules it will only make the change for you, but it probably makes more sense to track that information with the repository so everyone else does as well.

When we run git status at this point, Git will show us that we have “new commits” on the submodule.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: .gitmodules

modified: DbConnector (new commits)

no changes added to commit (use "git add" and/or "git commit -a")

If you set the configuration setting status.submodulesummary, Git will also show you a short summary of changes to your submodules:

$ git config status.submodulesummary 1

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: .gitmodules

modified: DbConnector (new commits)

Submodules changed but not updated:

\* DbConnector c3f01dc...c87d55d (4):

> catch non-null terminated lines

At this point if you run git diff we can see both that we have modified our .gitmodules file and also that there are a number of commits that we’ve pulled down and are ready to commit to our submodule project.

$ git diff

diff --git a/.gitmodules b/.gitmodules

index 6fc0b3d..fd1cc29 100644

--- a/.gitmodules

+++ b/.gitmodules

@@ -1,3 +1,4 @@

[submodule "DbConnector"]

path = DbConnector

url = https://github.com/chaconinc/DbConnector

+ branch = stable

Submodule DbConnector c3f01dc..c87d55d:

> catch non-null terminated lines

> more robust error handling

> more efficient db routine

> better connection routine

This is pretty cool as we can actually see the log of commits that we’re about to commit to in our submodule. Once committed, you can see this information after the fact as well when you run git log -p.

$ git log -p --submodule

commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae

Author: Scott Chacon <schacon@gmail.com>

Date: Wed Sep 17 16:37:02 2014 +0200

updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules

index 6fc0b3d..fd1cc29 100644

--- a/.gitmodules

+++ b/.gitmodules

@@ -1,3 +1,4 @@

[submodule "DbConnector"]

path = DbConnector

url = https://github.com/chaconinc/DbConnector

+ branch = stable

Submodule DbConnector c3f01dc..c87d55d:

> catch non-null terminated lines

> more robust error handling

> more efficient db routine

> better connection routine

Git will by default try to update **all** of your submodules when you run git submodule update --remote. If you have a lot of them, you may want to pass the name of just the submodule you want to try to update.

#### **Pulling Upstream Changes from the Project Remote**

Let’s now step into the shoes of your collaborator, who has their own local clone of the MainProject repository. Simply executing git pull to get your newly committed changes is not enough:

$ git pull

From https://github.com/chaconinc/MainProject

fb9093c..0a24cfc master -> origin/master

Fetching submodule DbConnector

From https://github.com/chaconinc/DbConnector

c3f01dc..c87d55d stable -> origin/stable

Updating fb9093c..0a24cfc

Fast-forward

.gitmodules | 2 +-

DbConnector | 2 +-

2 files changed, 2 insertions(+), 2 deletions(-)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: DbConnector (new commits)

Submodules changed but not updated:

\* DbConnector c87d55d...c3f01dc (4):

< catch non-null terminated lines

< more robust error handling

< more efficient db routine

< better connection routine

no changes added to commit (use "git add" and/or "git commit -a")

By default, the git pull command recursively fetches submodules changes, as we can see in the output of the first command above. However, it does not **update** the submodules. This is shown by the output of the git status command, which shows the submodule is “modified”, and has “new commits”. What’s more, the brackets showing the new commits point left (<), indicating that these commits are recorded in MainProject but are not present in the local DbConnector checkout. To finalize the update, you need to run git submodule update:

$ git submodule update --init --recursive

Submodule path 'vendor/plugins/demo': checked out '48679c6302815f6c76f1fe30625d795d9e55fc56'

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean

Note that to be on the safe side, you should run git submodule update with the --init flag in case the MainProject commits you just pulled added new submodules, and with the --recursive flag if any submodules have nested submodules.

If you want to automate this process, you can add the --recurse-submodules flag to the git pull command (since Git 2.14). This will make Git run git submodule update right after the pull, putting the submodules in the correct state. Moreover, if you want to make Git always pull with --recurse-submodules, you can set the configuration option submodule.recurse to true (this works for git pull since Git 2.15). This option will make Git use the --recurse-submodules flag for all commands that support it (except clone).

There is a special situation that can happen when pulling superproject updates: it could be that the upstream repository has changed the URL of the submodule in the .gitmodules file in one of the commits you pull. This can happen for example if the submodule project changes its hosting platform. In that case, it is possible for git pull --recurse-submodules, or git submodule update, to fail if the superproject references a submodule commit that is not found in the submodule remote locally configured in your repository. In order to remedy this situation, the git submodule sync command is required:

# copy the new URL to your local config

$ git submodule sync --recursive

# update the submodule from the new URL

$ git submodule update --init --recursive

#### **Working on a Submodule**

It’s quite likely that if you’re using submodules, you’re doing so because you really want to work on the code in the submodule at the same time as you’re working on the code in the main project (or across several submodules). Otherwise you would probably instead be using a simpler dependency management system (such as Maven or Rubygems).

So now let’s go through an example of making changes to the submodule at the same time as the main project and committing and publishing those changes at the same time.

So far, when we’ve run the git submodule update command to fetch changes from the submodule repositories, Git would get the changes and update the files in the subdirectory but will leave the sub-repository in what’s called a “detached HEAD” state. This means that there is no local working branch (like master, for example) tracking changes. With no working branch tracking changes, that means even if you commit changes to the submodule, those changes will quite possibly be lost the next time you run git submodule update. You have to do some extra steps if you want changes in a submodule to be tracked.

In order to set up your submodule to be easier to go in and hack on, you need to do two things. You need to go into each submodule and check out a branch to work on. Then you need to tell Git what to do if you have made changes and then git submodule update --remote pulls in new work from upstream. The options are that you can merge them into your local work, or you can try to rebase your local work on top of the new changes.

First of all, let’s go into our submodule directory and check out a branch.

$ cd DbConnector/

$ git checkout stable

Switched to branch 'stable'

Let’s try updating our submodule with the “merge” option. To specify it manually, we can just add the --merge option to our update call. Here we’ll see that there was a change on the server for this submodule and it gets merged in.

$ cd ..

$ git submodule update --remote --merge

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 4 (delta 2), reused 4 (delta 2)

Unpacking objects: 100% (4/4), done.

From https://github.com/chaconinc/DbConnector

c87d55d..92c7337 stable -> origin/stable

Updating c87d55d..92c7337

Fast-forward

src/main.c | 1 +

1 file changed, 1 insertion(+)

Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'

If we go into the DbConnector directory, we have the new changes already merged into our local stable branch. Now let’s see what happens when we make our own local change to the library and someone else pushes another change upstream at the same time.

$ cd DbConnector/

$ vim src/db.c

$ git commit -am 'Unicode support'

[stable f906e16] Unicode support

1 file changed, 1 insertion(+)

Now if we update our submodule we can see what happens when we have made a local change and upstream also has a change we need to incorporate.

$ cd ..

$ git submodule update --remote --rebase

First, rewinding head to replay your work on top of it...

Applying: Unicode support

Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'

If you forget the --rebase or --merge, Git will just update the submodule to whatever is on the server and reset your project to a detached HEAD state.

$ git submodule update --remote

Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'

If this happens, don’t worry, you can simply go back into the directory and check out your branch again (which will still contain your work) and merge or rebase origin/stable (or whatever remote branch you want) manually.

If you haven’t committed your changes in your submodule and you run a submodule update that would cause issues, Git will fetch the changes but not overwrite unsaved work in your submodule directory.

$ git submodule update --remote

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 4 (delta 0), reused 4 (delta 0)

Unpacking objects: 100% (4/4), done.

From https://github.com/chaconinc/DbConnector

5d60ef9..c75e92a stable -> origin/stable

error: Your local changes to the following files would be overwritten by checkout:

scripts/setup.sh

Please, commit your changes or stash them before you can switch branches.

Aborting

Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'

If you made changes that conflict with something changed upstream, Git will let you know when you run the update.

$ git submodule update --remote --merge

Auto-merging scripts/setup.sh

CONFLICT (content): Merge conflict in scripts/setup.sh

Recorded preimage for 'scripts/setup.sh'

Automatic merge failed; fix conflicts and then commit the result.

Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'

You can go into the submodule directory and fix the conflict just as you normally would.

#### **Publishing Submodule Changes**

Now we have some changes in our submodule directory. Some of these were brought in from upstream by our updates and others were made locally and aren’t available to anyone else yet as we haven’t pushed them yet.

$ git diff

Submodule DbConnector c87d55d..82d2ad3:

> Merge from origin/stable

> Update setup script

> Unicode support

> Remove unnecessary method

> Add new option for conn pooling

If we commit in the main project and push it up without pushing the submodule changes up as well, other people who try to check out our changes are going to be in trouble since they will have no way to get the submodule changes that are depended on. Those changes will only exist on our local copy.

In order to make sure this doesn’t happen, you can ask Git to check that all your submodules have been pushed properly before pushing the main project. The git push command takes the --recurse-submodules argument which can be set to either “check” or “on-demand”. The “check” option will make push simply fail if any of the committed submodule changes haven’t been pushed.

$ git push --recurse-submodules=check

The following submodule paths contain changes that can

not be found on any remote:

DbConnector

Please try

git push --recurse-submodules=on-demand

or cd to the path and use

git push

to push them to a remote.

As you can see, it also gives us some helpful advice on what we might want to do next. The simple option is to go into each submodule and manually push to the remotes to make sure they’re externally available and then try this push again. If you want the check behavior to happen for all pushes, you can make this behavior the default by doing git config push.recurseSubmodules check.

The other option is to use the “on-demand” value, which will try to do this for you.

$ git push --recurse-submodules=on-demand

Pushing submodule 'DbConnector'

Counting objects: 9, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (8/8), done.

Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.

Total 9 (delta 3), reused 0 (delta 0)

To https://github.com/chaconinc/DbConnector

c75e92a..82d2ad3 stable -> stable

Counting objects: 2, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.

Total 2 (delta 1), reused 0 (delta 0)

To https://github.com/chaconinc/MainProject

3d6d338..9a377d1 master -> master

As you can see there, Git went into the DbConnector module and pushed it before pushing the main project. If that submodule push fails for some reason, the main project push will also fail. You can make this behavior the default by doing git config push.recurseSubmodules on-demand.

#### **Merging Submodule Changes**

If you change a submodule reference at the same time as someone else, you may run into some problems. That is, if the submodule histories have diverged and are committed to diverging branches in a superproject, it may take a bit of work for you to fix.

If one of the commits is a direct ancestor of the other (a fast-forward merge), then Git will simply choose the latter for the merge, so that works fine.

Git will not attempt even a trivial merge for you, however. If the submodule commits diverge and need to be merged, you will get something that looks like this:

$ git pull

remote: Counting objects: 2, done.

remote: Compressing objects: 100% (1/1), done.

remote: Total 2 (delta 1), reused 2 (delta 1)

Unpacking objects: 100% (2/2), done.

From https://github.com/chaconinc/MainProject

9a377d1..eb974f8 master -> origin/master

Fetching submodule DbConnector

warning: Failed to merge submodule DbConnector (merge following commits not found)

Auto-merging DbConnector

CONFLICT (submodule): Merge conflict in DbConnector

Automatic merge failed; fix conflicts and then commit the result.

So basically what has happened here is that Git has figured out that the two branches record points in the submodule’s history that are divergent and need to be merged. It explains it as “merge following commits not found”, which is confusing but we’ll explain why that is in a bit.

To solve the problem, you need to figure out what state the submodule should be in. Strangely, Git doesn’t really give you much information to help out here, not even the SHA-1s of the commits of both sides of the history. Fortunately, it’s simple to figure out. If you run git diff you can get the SHA-1s of the commits recorded in both branches you were trying to merge.

$ git diff

diff --cc DbConnector

index eb41d76,c771610..0000000

--- a/DbConnector

+++ b/DbConnector

So, in this case, eb41d76 is the commit in our submodule that **we** had and c771610 is the commit that upstream had. If we go into our submodule directory, it should already be on eb41d76 as the merge would not have touched it. If for whatever reason it’s not, you can simply create and checkout a branch pointing to it.

What is important is the SHA-1 of the commit from the other side. This is what you’ll have to merge in and resolve. You can either just try the merge with the SHA-1 directly, or you can create a branch for it and then try to merge that in. We would suggest the latter, even if only to make a nicer merge commit message.

So, we will go into our submodule directory, create a branch named “try-merge” based on that second SHA-1 from git diff, and manually merge.

$ cd DbConnector

$ git rev-parse HEAD

eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610

$ git merge try-merge

Auto-merging src/main.c

CONFLICT (content): Merge conflict in src/main.c

Recorded preimage for 'src/main.c'

Automatic merge failed; fix conflicts and then commit the result.

We got an actual merge conflict here, so if we resolve that and commit it, then we can simply update the main project with the result.

$ vim src/main.c **(1)**

$ git add src/main.c

$ git commit -am 'merged our changes'

Recorded resolution for 'src/main.c'.

[master 9fd905e] merged our changes

$ cd .. **(2)**

$ git diff **(3)**

diff --cc DbConnector

index eb41d76,c771610..0000000

--- a/DbConnector

+++ b/DbConnector

@@@ -1,1 -1,1 +1,1 @@@

- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135

-Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d

++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a

$ git add DbConnector **(4)**

$ git commit -m "Merge Tom's Changes" **(5)**

[master 10d2c60] Merge Tom's Changes

1. First we resolve the conflict.
2. Then we go back to the main project directory.
3. We can check the SHA-1s again.
4. Resolve the conflicted submodule entry.
5. Commit our merge.

It can be a bit confusing, but it’s really not very hard.

Interestingly, there is another case that Git handles. If a merge commit exists in the submodule directory that contains **both** commits in its history, Git will suggest it to you as a possible solution. It sees that at some point in the submodule project, someone merged branches containing these two commits, so maybe you’ll want that one.

This is why the error message from before was “merge following commits not found”, because it could not do **this**. It’s confusing because who would expect it to **try** to do this?

If it does find a single acceptable merge commit, you’ll see something like this:

$ git merge origin/master

warning: Failed to merge submodule DbConnector (not fast-forward)

Found a possible merge resolution for the submodule:

9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes

If this is correct simply add it to the index for example

by using:

git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a "DbConnector"

which will accept this suggestion.

Auto-merging DbConnector

CONFLICT (submodule): Merge conflict in DbConnector

Automatic merge failed; fix conflicts and then commit the result.

The suggested command Git is providing will update the index as though you had run git add (which clears the conflict), then commit. You probably shouldn’t do this though. You can just as easily go into the submodule directory, see what the difference is, fast-forward to this commit, test it properly, and then commit it.

$ cd DbConnector/

$ git merge 9fd905e

Updating eb41d76..9fd905e

Fast-forward

$ cd ..

$ git add DbConnector

$ git commit -am 'Fast forward to a common submodule child'

This accomplishes the same thing, but at least this way you can verify that it works and you have the code in your submodule directory when you’re done.

### **Submodule Tips**

There are a few things you can do to make working with submodules a little easier.

#### **Submodule Foreach**

There is a foreach submodule command to run some arbitrary command in each submodule. This can be really helpful if you have a number of submodules in the same project.

For example, let’s say we want to start a new feature or do a bugfix and we have work going on in several submodules. We can easily stash all the work in all our submodules.

$ git submodule foreach 'git stash'

Entering 'CryptoLibrary'

No local changes to save

Entering 'DbConnector'

Saved working directory and index state WIP on stable: 82d2ad3 Merge from origin/stable

HEAD is now at 82d2ad3 Merge from origin/stable

Then we can create a new branch and switch to it in all our submodules.

$ git submodule foreach 'git checkout -b featureA'

Entering 'CryptoLibrary'

Switched to a new branch 'featureA'

Entering 'DbConnector'

Switched to a new branch 'featureA'

You get the idea. One really useful thing you can do is produce a nice unified diff of what is changed in your main project and all your subprojects as well.

$ git diff; git submodule foreach 'git diff'

Submodule DbConnector contains modified content

diff --git a/src/main.c b/src/main.c

index 210f1ae..1f0acdc 100644

--- a/src/main.c

+++ b/src/main.c

@@ -245,6 +245,8 @@ static int handle\_alias(int \*argcp, const char \*\*\*argv)

commit\_pager\_choice();

+ url = url\_decode(url\_orig);

+

/\* build alias\_argv \*/

alias\_argv = xmalloc(sizeof(\*alias\_argv) \* (argc + 1));

alias\_argv[0] = alias\_string + 1;

Entering 'DbConnector'

diff --git a/src/db.c b/src/db.c

index 1aaefb6..5297645 100644

--- a/src/db.c

+++ b/src/db.c

@@ -93,6 +93,11 @@ char \*url\_decode\_mem(const char \*url, int len)

return url\_decode\_internal(&url, len, NULL, &out, 0);

}

+char \*url\_decode(const char \*url)

+{

+ return url\_decode\_mem(url, strlen(url));

+}

+

char \*url\_decode\_parameter\_name(const char \*\*query)

{

struct strbuf out = STRBUF\_INIT;

Here we can see that we’re defining a function in a submodule and calling it in the main project. This is obviously a simplified example, but hopefully it gives you an idea of how this may be useful.

#### **Useful Aliases**

You may want to set up some aliases for some of these commands as they can be quite long and you can’t set configuration options for most of them to make them defaults. We covered setting up Git aliases in [Git Aliases](https://git-scm.com/book/en/v2/ch00/_git_aliases), but here is an example of what you may want to set up if you plan on working with submodules in Git a lot.

$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"

$ git config alias.spush 'push --recurse-submodules=on-demand'

$ git config alias.supdate 'submodule update --remote --merge'

This way you can simply run git supdate when you want to update your submodules, or git spush to push with submodule dependency checking.

### **Issues with Submodules**

Using submodules isn’t without hiccups, however.

#### **Switching branches**

For instance, switching branches with submodules in them can also be tricky with Git versions older than Git 2.13. If you create a new branch, add a submodule there, and then switch back to a branch without that submodule, you still have the submodule directory as an untracked directory:

$ git --version

git version 2.12.2

$ git checkout -b add-crypto

Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary

Cloning into 'CryptoLibrary'...

...

$ git commit -am 'Add crypto library'

[add-crypto 4445836] Add crypto library

2 files changed, 4 insertions(+)

create mode 160000 CryptoLibrary

$ git checkout master

warning: unable to rmdir CryptoLibrary: Directory not empty

Switched to branch 'master'

Your branch is up-to-date with 'origin/master'.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

CryptoLibrary/

nothing added to commit but untracked files present (use "git add" to track)

Removing the directory isn’t difficult, but it can be a bit confusing to have that in there. If you do remove it and then switch back to the branch that has that submodule, you will need to run submodule update --init to repopulate it.

$ git clean -ffdx

Removing CryptoLibrary/

$ git checkout add-crypto

Switched to branch 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init

Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/

Makefile includes scripts src

Again, not really very difficult, but it can be a little confusing.

Newer Git versions (Git >= 2.13) simplify all this by adding the --recurse-submodules flag to the git checkout command, which takes care of placing the submodules in the right state for the branch we are switching to.

$ git --version

git version 2.13.3

$ git checkout -b add-crypto

Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary

Cloning into 'CryptoLibrary'...

...

$ git commit -am 'Add crypto library'

[add-crypto 4445836] Add crypto library

2 files changed, 4 insertions(+)

create mode 160000 CryptoLibrary

$ git checkout --recurse-submodules master

Switched to branch 'master'

Your branch is up-to-date with 'origin/master'.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean

Using the --recurse-submodules flag of git checkout can also be useful when you work on several branches in the superproject, each having your submodule pointing at different commits. Indeed, if you switch between branches that record the submodule at different commits, upon executing git status the submodule will appear as “modified”, and indicate “new commits”. That is because the submodule state is by default not carried over when switching branches.

This can be really confusing, so it’s a good idea to always git checkout --recurse-submodules when your project has submodules. For older Git versions that do not have the --recurse-submodules flag, after the checkout you can use git submodule update --init --recursive to put the submodules in the right state.

Luckily, you can tell Git (>=2.14) to always use the --recurse-submodules flag by setting the configuration option submodule.recurse: git config submodule.recurse true. As noted above, this will also make Git recurse into submodules for every command that has a --recurse-submodules option (except git clone).

#### **Switching from subdirectories to submodules**

The other main caveat that many people run into involves switching from subdirectories to submodules. If you’ve been tracking files in your project and you want to move them out into a submodule, you must be careful or Git will get angry at you. Assume that you have files in a subdirectory of your project, and you want to switch it to a submodule. If you delete the subdirectory and then run submodule add, Git yells at you:

$ rm -Rf CryptoLibrary/

$ git submodule add https://github.com/chaconinc/CryptoLibrary

'CryptoLibrary' already exists in the index

You have to unstage the CryptoLibrary directory first. Then you can add the submodule:

$ git rm -r CryptoLibrary

$ git submodule add https://github.com/chaconinc/CryptoLibrary

Cloning into 'CryptoLibrary'...

remote: Counting objects: 11, done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 11 (delta 0), reused 11 (delta 0)

Unpacking objects: 100% (11/11), done.

Checking connectivity... done.

Now suppose you did that in a branch. If you try to switch back to a branch where those files are still in the actual tree rather than a submodule – you get this error:

$ git checkout master

error: The following untracked working tree files would be overwritten by checkout:

CryptoLibrary/Makefile

CryptoLibrary/includes/crypto.h

...

Please move or remove them before you can switch branches.

Aborting

You can force it to switch with checkout -f, but be careful that you don’t have unsaved changes in there as they could be overwritten with that command.

$ git checkout -f master

warning: unable to rmdir CryptoLibrary: Directory not empty

Switched to branch 'master'

Then, when you switch back, you get an empty CryptoLibrary directory for some reason and git submodule update may not fix it either. You may need to go into your submodule directory and run a git checkout . to get all your files back. You could run this in a submodule foreach script to run it for multiple submodules.

It’s important to note that submodules these days keep all their Git data in the top project’s .git directory, so unlike much older versions of Git, destroying a submodule directory won’t lose any commits or branches that you had.

With these tools, submodules can be a fairly simple and effective method for developing on several related but still separate projects simultaneously.

**Subtree Merging**

Now that you’ve seen the difficulties of the submodule system, let’s look at an alternate way to solve the same problem. When Git merges, it looks at what it has to merge together and then chooses an appropriate merging strategy to use. If you’re merging two branches, Git uses a *recursive* strategy. If you’re merging more than two branches, Git picks the *octopus* strategy. These strategies are automatically chosen for you because the recursive strategy can handle complex three-way merge situations — for example, more than one common ancestor — but it can only handle merging two branches. The octopus merge can handle multiple branches but is more cautious to avoid difficult conflicts, so it’s chosen as the default strategy if you’re trying to merge more than two branches.

However, there are other strategies you can choose as well. One of them is the *subtree* merge, and you can use it to deal with the subproject issue. Here you’ll see how to do the same rack embedding as in the last section, but using subtree merges instead.

The idea of the subtree merge is that you have two projects, and one of the projects maps to a subdirectory of the other one and vice versa. When you specify a subtree merge, Git is smart enough to figure out that one is a subtree of the other and merge appropriately — it’s pretty amazing.

You first add the Rack application to your project. You add the Rack project as a remote reference in your own project and then check it out into its own branch:

$ git remote add rack\_remote git@github.com:schacon/rack.git

$ git fetch rack\_remote

warning: no common commits

remote: Counting objects: 3184, done.

remote: Compressing objects: 100% (1465/1465), done.

remote: Total 3184 (delta 1952), reused 2770 (delta 1675)

Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.

Resolving deltas: 100% (1952/1952), done.

From git@github.com:schacon/rack

\* [new branch] build -> rack\_remote/build

\* [new branch] master -> rack\_remote/master

\* [new branch] rack-0.4 -> rack\_remote/rack-0.4

\* [new branch] rack-0.9 -> rack\_remote/rack-0.9

$ git checkout -b rack\_branch rack\_remote/master

Branch rack\_branch set up to track remote branch refs/remotes/rack\_remote/master.

Switched to a new branch "rack\_branch"

Now you have the root of the Rack project in your rack\_branch branch and your own project in the master branch. If you check out one and then the other, you can see that they have different project roots:

$ ls

AUTHORS KNOWN-ISSUES Rakefile contrib lib

COPYING README bin example test

$ git checkout master

Switched to branch "master"

$ ls

README

You want to pull the Rack project into your master project as a subdirectory. You can do that in Git with git read-tree. You’ll learn more about read-tree and its friends in Chapter 9, but for now know that it reads the root tree of one branch into your current staging area and working directory. You just switched back to your master branch, and you pull the rack branch into the rack subdirectory of your master branch of your main project:

$ git read-tree --prefix=rack/ -u rack\_branch

When you commit, it looks like you have all the Rack files under that subdirectory — as though you copied them in from a tarball. What gets interesting is that you can fairly easily merge changes from one of the branches to the other. So, if the Rack project updates, you can pull in upstream changes by switching to that branch and pulling:

$ git checkout rack\_branch

$ git pull

Then, you can merge those changes back into your master branch. You can use git merge -s subtree and it will work fine; but Git will also merge the histories together, which you probably don’t want. To pull in the changes and prepopulate the commit message, use the --squash and --no-commit options as well as the -s subtree strategy option:

$ git checkout master

$ git merge --squash -s subtree --no-commit rack\_branch

Squash commit -- not updating HEAD

Automatic merge went well; stopped before committing as requested

All the changes from your Rack project are merged in and ready to be committed locally. You can also do the opposite — make changes in the rack subdirectory of your master branch and then merge them into your rack\_branch branch later to submit them to the maintainers or push them upstream.

To get a diff between what you have in your rack subdirectory and the code in your rack\_branch branch — to see if you need to merge them — you can’t use the normal diff command. Instead, you must run git diff-tree with the branch you want to compare to:

$ git diff-tree -p rack\_branch

Or, to compare what is in your rack subdirectory with what the master branch on the server was the last time you fetched, you can run

$ git diff-tree -p rack\_remote/master

**Git Configuration**

Git configuration settings with the git config command. One of the first things you did was set up your name and e-mail address:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

Now you’ll learn a few of the more interesting options that you can set in this manner to customize your Git usage.

You saw some simple Git configuration details in the first chapter, but I’ll go over them again quickly here. Git uses a series of configuration files to determine non-default behavior that you may want. The first place Git looks for these values is in an /etc/gitconfig file, which contains values for every user on the system and all of their repositories. If you pass the option --system to git config, it reads and writes from this file specifically.

The next place Git looks is the ~/.gitconfig file, which is specific to each user. You can make Git read and write to this file by passing the --global option.

Finally, Git looks for configuration values in the config file in the Git directory (.git/config) of whatever repository you’re currently using. These values are specific to that single repository. Each level overwrites values in the previous level, so values in .git/config trump those in /etc/gitconfig, for instance. You can also set these values by manually editing the file and inserting the correct syntax, but it’s generally easier to run the git config command.

## 

## **Basic Client Configuration**

The configuration options recognized by Git fall into two categories: client side and server side. The majority of the options are client side—configuring your personal working preferences. Although tons of options are available, I’ll only cover the few that either are commonly used or can significantly affect your workflow. Many options are useful only in edge cases that I won’t go over here. If you want to see a list of all the options your version of Git recognizes, you can run

$ git config --help

The manual page for git config lists all the available options in quite a bit of detail.

### **core.editor**

By default, Git uses whatever you’ve set as your default text editor or else falls back to the Vi editor to create and edit your commit and tag messages. To change that default to something else, you can use the core.editor setting:

$ git config --global core.editor emacs

Now, no matter what is set as your default shell editor variable, Git will fire up Emacs to edit messages.

### **commit.template**

If you set this to the path of a file on your system, Git will use that file as the default message when you commit. For instance, suppose you create a template file at $HOME/.gitmessage.txt that looks like this:

subject line

what happened

[ticket: X]

To tell Git to use it as the default message that appears in your editor when you run git commit, set the commit.template configuration value:

$ git config --global commit.template $HOME/.gitmessage.txt

$ git commit

Then, your editor will open to something like this for your placeholder commit message when you commit:

subject line

what happened

[ticket: X]

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: lib/test.rb

#

~

~

".git/COMMIT\_EDITMSG" 14L, 297C

If you have a commit-message policy in place, then putting a template for that policy on your system and configuring Git to use it by default can help increase the chance of that policy being followed regularly.

### **core.pager**

The core.pager setting determines what pager is used when Git pages output such as log and diff. You can set it to more or to your favorite pager (by default, it’s less), or you can turn it off by setting it to a blank string:

$ git config --global core.pager ''

If you run that, Git will page the entire output of all commands, no matter how long it is.

### **user.signingkey**

If you’re making signed annotated tags (as discussed in Chapter 2), setting your GPG signing key as a configuration setting makes things easier. Set your key ID like so:

$ git config --global user.signingkey <gpg-key-id>

Now, you can sign tags without having to specify your key every time with the git tag command:

$ git tag -s <tag-name>

### **core.excludesfile**

You can put patterns in your project’s .gitignore file to have Git not see them as untracked files or try to stage them when you run git add on them, as discussed in Chapter 2. However, if you want another file outside of your project to hold those values or have extra values, you can tell Git where that file is with the core.excludesfile setting. Simply set it to the path of a file that has content similar to what a .gitignore file would have.

### **help.autocorrect**

This option is available only in Git 1.6.1 and later. If you mistype a command in Git, it shows you something like this:

$ git com

git: 'com' is not a git-command. See 'git --help'.

Did you mean this?

commit

If you set help.autocorrect to 1, Git will automatically run the command if it has only one match under this scenario.

## **Colors in Git**

Git can color its output to your terminal, which can help you visually parse the output quickly and easily. A number of options can help you set the coloring to your preference.

### **color.ui**

Git automatically colors most of its output if you ask it to. You can get very specific about what you want colored and how; but to turn on all the default terminal coloring, set color.ui to true:

$ git config --global color.ui true

When that value is set, Git colors its output if the output goes to a terminal. Other possible settings are false, which never colors the output, and always, which sets colors all the time, even if you’re redirecting Git commands to a file or piping them to another command.

You’ll rarely want color.ui = always. In most scenarios, if you want color codes in your redirected output, you can instead pass a --color flag to the Git command to force it to use color codes. The color.ui = true setting is almost always what you’ll want to use.

### **color.\***

If you want to be more specific about which commands are colored and how, Git provides verb-specific coloring settings. Each of these can be set to true, false, or always:

color.branch

color.diff

color.interactive

color.status

In addition, each of these has subsettings you can use to set specific colors for parts of the output, if you want to override each color. For example, to set the meta information in your diff output to blue foreground, black background, and bold text, you can run

$ git config --global color.diff.meta "blue black bold"

You can set the color to any of the following values: normal, black, red, green, yellow, blue, magenta, cyan, or white. If you want an attribute like bold in the previous example, you can choose from bold, dim, ul, blink, and reverse.

See the git config manpage for all the subsettings you can configure, if you want to do that.

## **External Merge and Diff Tools**

Although Git has an internal implementation of diff, which is what you’ve been using, you can set up an external tool instead. You can also set up a graphical merge conflict-resolution tool instead of having to resolve conflicts manually. I’ll demonstrate setting up the Perforce Visual Merge Tool (P4Merge) to do your diffs and merge resolutions, because it’s a nice graphical tool and it’s free.

If you want to try this out, P4Merge works on all major platforms, so you should be able to do so. I’ll use path names in the examples that work on Mac and Linux systems; for Windows, you’ll have to change /usr/local/bin to an executable path in your environment.

You can download P4Merge here:

http://www.perforce.com/perforce/downloads/component.html

To begin, you’ll set up external wrapper scripts to run your commands. I’ll use the Mac path for the executable; in other systems, it will be where your p4merge binary is installed. Set up a merge wrapper script named extMerge that calls your binary with all the arguments provided:

$ cat /usr/local/bin/extMerge

#!/bin/sh

/Applications/p4merge.app/Contents/MacOS/p4merge $\*

The diff wrapper checks to make sure seven arguments are provided and passes two of them to your merge script. By default, Git passes the following arguments to the diff program:

path old-file old-hex old-mode new-file new-hex new-mode

Because you only want the old-file and new-file arguments, you use the wrapper script to pass the ones you need.

$ cat /usr/local/bin/extDiff

#!/bin/sh

[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"

You also need to make sure these tools are executable:

$ sudo chmod +x /usr/local/bin/extMerge

$ sudo chmod +x /usr/local/bin/extDiff

Now you can set up your config file to use your custom merge resolution and diff tools. This takes a number of custom settings: merge.tool to tell Git what strategy to use, mergetool.\*.cmd to specify how to run the command, mergetool.trustExitCode to tell Git if the exit code of that program indicates a successful merge resolution or not, and diff.external to tell Git what command to run for diffs. So, you can either run four config commands

$ git config --global merge.tool extMerge

$ git config --global mergetool.extMerge.cmd \

'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'

$ git config --global mergetool.trustExitCode false

$ git config --global diff.external extDiff

or you can edit your ~/.gitconfig file to add these lines:

[merge]

tool = extMerge

[mergetool "extMerge"]

cmd = extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"

trustExitCode = false

[diff]

external = extDiff

After all this is set, if you run diff commands such as this:

$ git diff 32d1776b1^ 32d1776b1

Instead of getting the diff output on the command line, Git fires up P4Merge, which looks something like Figure 7-1.

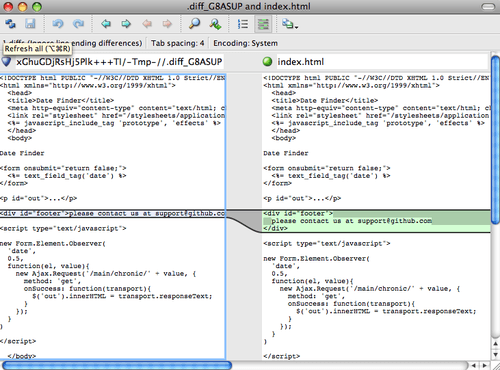


Figure 7-1. P4Merge.

If you try to merge two branches and subsequently have merge conflicts, you can run the command git mergetool; it starts P4Merge to let you resolve the conflicts through that GUI tool.

The nice thing about this wrapper setup is that you can change your diff and merge tools easily. For example, to change your extDiff and extMerge tools to run the KDiff3 tool instead, all you have to do is edit your extMerge file:

$ cat /usr/local/bin/extMerge

#!/bin/sh

/Applications/kdiff3.app/Contents/MacOS/kdiff3 $\*

Now, Git will use the KDiff3 tool for diff viewing and merge conflict resolution.

Git comes preset to use a number of other merge-resolution tools without your having to set up the cmd configuration. You can set your merge tool to kdiff3, opendiff, tkdiff, meld, xxdiff, emerge, vimdiff, or gvimdiff. If you’re not interested in using KDiff3 for diff but rather want to use it just for merge resolution, and the kdiff3 command is in your path, then you can run

$ git config --global merge.tool kdiff3

If you run this instead of setting up the extMerge and extDiff files, Git will use KDiff3 for merge resolution and the normal Git diff tool for diffs.

## **Formatting and Whitespace**

Formatting and whitespace issues are some of the more frustrating and subtle problems that many developers encounter when collaborating, especially cross-platform. It’s very easy for patches or other collaborated work to introduce subtle whitespace changes because editors silently introduce them or Windows programmers add carriage returns at the end of lines they touch in cross-platform projects. Git has a few configuration options to help with these issues.

### **core.autocrlf**

If you’re programming on Windows or using another system but working with people who are programming on Windows, you’ll probably run into line-ending issues at some point. This is because Windows uses both a carriage-return character and a linefeed character for newlines in its files, whereas Mac and Linux systems use only the linefeed character. This is a subtle but incredibly annoying fact of cross-platform work.

Git can handle this by auto-converting CRLF line endings into LF when you commit, and vice versa when it checks out code onto your filesystem. You can turn on this functionality with the core.autocrlf setting. If you’re on a Windows machine, set it to true — this converts LF endings into CRLF when you check out code:

$ git config --global core.autocrlf true

If you’re on a Linux or Mac system that uses LF line endings, then you don’t want Git to automatically convert them when you check out files; however, if a file with CRLF endings accidentally gets introduced, then you may want Git to fix it. You can tell Git to convert CRLF to LF on commit but not the other way around by setting core.autocrlf to input:

$ git config --global core.autocrlf input

This setup should leave you with CRLF endings in Windows checkouts but LF endings on Mac and Linux systems and in the repository.

If you’re a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to false:

$ git config --global core.autocrlf false

### **core.whitespace**

Git comes preset to detect and fix some whitespace issues. It can look for four primary whitespace issues — two are enabled by default and can be turned off, and two aren’t enabled by default but can be activated.

The two that are turned on by default are trailing-space, which looks for spaces at the end of a line, and space-before-tab, which looks for spaces before tabs at the beginning of a line.

The two that are disabled by default but can be turned on are indent-with-non-tab, which looks for lines that begin with eight or more spaces instead of tabs, and cr-at-eol, which tells Git that carriage returns at the end of lines are OK.

You can tell Git which of these you want enabled by setting core.whitespace to the values you want on or off, separated by commas. You can disable settings by either leaving them out of the setting string or prepending a - in front of the value. For example, if you want all but cr-at-eol to be set, you can do this:

$ git config --global core.whitespace \

trailing-space,space-before-tab,indent-with-non-tab

Git will detect these issues when you run a git diff command and try to color them so you can possibly fix them before you commit. It will also use these values to help you when you apply patches with git apply. When you’re applying patches, you can ask Git to warn you if it’s applying patches with the specified whitespace issues:

$ git apply --whitespace=warn <patch>

Or you can have Git try to automatically fix the issue before applying the patch:

$ git apply --whitespace=fix <patch>

These options apply to the git rebase command as well. If you’ve committed whitespace issues but haven’t yet pushed upstream, you can run a rebase with the --whitespace=fix option to have Git automatically fix whitespace issues as it’s rewriting the patches.

## **Server Configuration**

Not nearly as many configuration options are available for the server side of Git, but there are a few interesting ones you may want to take note of.

### **receive.fsckObjects**

By default, Git doesn’t check for consistency all the objects it receives during a push. Although Git can check to make sure each object still matches its SHA-1 checksum and points to valid objects, it doesn’t do that by default on every push. This is a relatively expensive operation and may add a lot of time to each push, depending on the size of the repository or the push. If you want Git to check object consistency on every push, you can force it to do so by setting receive.fsckObjects to true:

$ git config --system receive.fsckObjects true

Now, Git will check the integrity of your repository before each push is accepted to make sure faulty clients aren’t introducing corrupt data.

### **receive.denyNonFastForwards**

If you rebase commits that you’ve already pushed and then try to push again, or otherwise try to push a commit to a remote branch that doesn’t contain the commit that the remote branch currently points to, you’ll be denied. This is generally good policy; but in the case of the rebase, you may determine that you know what you’re doing and can force-update the remote branch with a -f flag to your push command.

To disable the ability to force-update remote branches to non-fast-forward references, set receive.denyNonFastForwards:

$ git config --system receive.denyNonFastForwards true

The other way you can do this is via server-side receive hooks, which I’ll cover in a bit. That approach lets you do more complex things like deny non-fast-forwards to a certain subset of users.

### **receive.denyDeletes**

One of the workarounds to the denyNonFastForwards policy is for the user to delete the branch and then push it back up with the new reference. In newer versions of Git (beginning with version 1.6.1), you can set receive.denyDeletes to true:

$ git config --system receive.denyDeletes true

This denies branch and tag deletion over a push across the board — no user can do it. To remove remote branches, you must remove the ref files from the server manually

**Git Attributes**

Some of these settings can also be specified for a path, so that Git applies those settings only for a subdirectory or subset of files. These path-specific settings are called Git attributes and are set either in a .gitattributes file in one of your directories (normally the root of your project) or in the .git/info/attributes file if you don’t want the attributes file committed with your project.

Using attributes, you can do things like specify separate merge strategies for individual files or directories in your project, tell Git how to diff non-text files, or have Git filter content before you check it into or out of Git. In this section, you’ll learn about some of the attributes you can set on your paths in your Git project and see a few examples of using this feature in practice.

## **Binary Files**

One cool trick for which you can use Git attributes is telling Git which files are binary (in cases it otherwise may not be able to figure out) and giving Git special instructions about how to handle those files. For instance, some text files may be machine generated and not diffable, whereas some binary files can be diffed — you’ll see how to tell Git which is which.

### **Identifying Binary Files**

Some files look like text files but for all intents and purposes are to be treated as binary data. For instance, Xcode projects on the Mac contain a file that ends in .pbxproj, which is basically a JSON (plain text javascript data format) dataset written out to disk by the IDE that records your build settings and so on. Although it’s technically a text file, because it’s all ASCII, you don’t want to treat it as such because it’s really a lightweight database — you can’t merge the contents if two people changed it, and diffs generally aren’t helpful. The file is meant to be consumed by a machine. In essence, you want to treat it like a binary file.

To tell Git to treat all pbxproj files as binary data, add the following line to your .gitattributes file:

\*.pbxproj -crlf -diff

Now, Git won’t try to convert or fix CRLF issues; nor will it try to compute or print a diff for changes in this file when you run git show or git diff on your project. You can also use a built-in macro binary that means -crlf -diff:

\*.pbxproj binary

### **Diffing Binary Files**

In Git, you can use the attributes functionality to effectively diff binary files. You do this by telling Git how to convert your binary data to a text format that can be compared via the normal diff. But the question is how do you convert *binary* data to a text? The best solution is to find some tool that does conversion for your binary format to a text representation. Unfortunately, very few binary formats can be represented as human readable text (imagine trying to convert audio data to a text). If this is the case and you failed to get a text presentation of your file's contents, it's often relatively easy to get a human readable description of that content, or metadata. Metadata won't give you a full representation of your file's content, but in any case it's better than nothing.

We'll make use of the both described approaches to get usable diffs for some widely used binary formats.

Side note: There are different kinds of binary formats with a text content, which are hard to find usable converter for. In such a case you could try to extract a text from your file with the strings program. Some of these files may use an UTF-16 encoding or other "codepages" and strings won’t find anything useful in there. Your mileage may vary. However, strings is available on most Mac and Linux systems, so it may be a good first try to do this with many binary formats.

#### **MS Word files**

First, you’ll use the described technique to solve one of the most annoying problems known to humanity: version-controlling Word documents. Everyone knows that Word is the most horrific editor around; but, oddly, everyone uses it. If you want to version-control Word documents, you can stick them in a Git repository and commit every once in a while; but what good does that do? If you run git diff normally, you only see something like this:

$ git diff

diff --git a/chapter1.doc b/chapter1.doc

index 88839c4..4afcb7c 100644

Binary files a/chapter1.doc and b/chapter1.doc differ

You can’t directly compare two versions unless you check them out and scan them manually, right? It turns out you can do this fairly well using Git attributes. Put the following line in your .gitattributes file:

\*.doc diff=word

This tells Git that any file that matches this pattern (.doc) should use the "word" filter when you try to view a diff that contains changes. What is the "word" filter? You have to set it up. Here you’ll configure Git to use the catdoc program, which was written specifically for extracting text from a binary MS Word documents (you can get it from http://www.wagner.pp.ru/~vitus/software/catdoc/), to convert Word documents into readable text files, which it will then diff properly:

$ git config diff.word.textconv catdoc

This command adds a section to your .git/config that looks like this:

[diff "word"]

textconv = catdoc

Now Git knows that if it tries to do a diff between two snapshots, and any of the files end in .doc, it should run those files through the "word" filter, which is defined as the catdoc program. This effectively makes nice text-based versions of your Word files before attempting to diff them.

Here’s an example. I put Chapter 1 of this book into Git, added some text to a paragraph, and saved the document. Then, I ran git diff to see what changed:

$ git diff

diff --git a/chapter1.doc b/chapter1.doc

index c1c8a0a..b93c9e4 100644

--- a/chapter1.doc

+++ b/chapter1.doc

@@ -128,7 +128,7 @@ and data size)

Since its birth in 2005, Git has evolved and matured to be easy to use

and yet retain these initial qualities. It’s incredibly fast, it’s

very efficient with large projects, and it has an incredible branching

-system for non-linear development.

+system for non-linear development (See Chapter 3).

Git successfully and succinctly tells me that I added the string "(See Chapter 3)", which is correct. Works perfectly!

#### **OpenDocument Text files**

The same approach that we used for MS Word files (\*.doc) can be used for OpenDocument Text files (\*.odt) created by OpenOffice.org.

Add the following line to your .gitattributes file:

\*.odt diff=odt

Now set up the odt diff filter in .git/config:

[diff "odt"]

binary = true

textconv = /usr/local/bin/odt-to-txt

OpenDocument files are actually zip’ped directories containing multiple files (the content in an XML format, stylesheets, images, etc.). We’ll need to write a script to extract the content and return it as plain text. Create a file /usr/local/bin/odt-to-txt (you are free to put it into a different directory) with the following content:

#! /usr/bin/env perl

# Simplistic OpenDocument Text (.odt) to plain text converter.

# Author: Philipp Kempgen

if (! defined($ARGV[0])) {

print STDERR "No filename given!\n";

print STDERR "Usage: $0 filename\n";

exit 1;

}

my $content = '';

open my $fh, '-|', 'unzip', '-qq', '-p', $ARGV[0], 'content.xml' or die $!;

{

local $/ = undef; # slurp mode

$content = <$fh>;

}

close $fh;

$\_ = $content;

s/<text:span\b[^>]\*>//g; # remove spans

s/<text:h\b[^>]\*>/\n\n\*\*\*\*\* /g; # headers

s/<text:list-item\b[^>]\*>\s\*<text:p\b[^>]\*>/\n -- /g; # list items

s/<text:list\b[^>]\*>/\n\n/g; # lists

s/<text:p\b[^>]\*>/\n /g; # paragraphs

s/<[^>]+>//g; # remove all XML tags

s/\n{2,}/\n\n/g; # remove multiple blank lines

s/\A\n+//; # remove leading blank lines

print "\n", $\_, "\n\n";

And make it executable

chmod +x /usr/local/bin/odt-to-txt

Now git diff will be able to tell you what changed in .odt files.

#### **Image files**

Another interesting problem you can solve this way involves diffing image files. One way to do this is to run PNG files through a filter that extracts their EXIF information — metadata that is recorded with most image formats. If you download and install the exiftool program, you can use it to convert your images into text about the metadata, so at least the diff will show you a textual representation of any changes that happened:

$ echo '\*.png diff=exif' >> .gitattributes

$ git config diff.exif.textconv exiftool

If you replace an image in your project and run git diff, you see something like this:

diff --git a/image.png b/image.png

index 88839c4..4afcb7c 100644

--- a/image.png

+++ b/image.png

@@ -1,12 +1,12 @@

ExifTool Version Number : 7.74

-File Size : 70 kB

-File Modification Date/Time : 2009:04:17 10:12:35-07:00

+File Size : 94 kB

+File Modification Date/Time : 2009:04:21 07:02:43-07:00

File Type : PNG

MIME Type : image/png

-Image Width : 1058

-Image Height : 889

+Image Width : 1056

+Image Height : 827

Bit Depth : 8

Color Type : RGB with Alpha

You can easily see that the file size and image dimensions have both changed.

## **Keyword Expansion**

SVN- or CVS-style keyword expansion is often requested by developers used to those systems. The main problem with this in Git is that you can’t modify a file with information about the commit after you’ve committed, because Git checksums the file first. However, you can inject text into a file when it’s checked out and remove it again before it’s added to a commit. Git attributes offers you two ways to do this.

First, you can inject the SHA-1 checksum of a blob into an $Id$ field in the file automatically. If you set this attribute on a file or set of files, then the next time you check out that branch, Git will replace that field with the SHA-1 of the blob. It’s important to notice that it isn’t the SHA of the commit, but of the blob itself:

$ echo '\*.txt ident' >> .gitattributes

$ echo '$Id$' > test.txt

The next time you check out this file, Git injects the SHA of the blob:

$ rm test.txt

$ git checkout -- test.txt

$ cat test.txt

$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $

However, that result is of limited use. If you’ve used keyword substitution in CVS or Subversion, you can include a datestamp — the SHA isn’t all that helpful, because it’s fairly random and you can’t tell if one SHA is older or newer than another.

It turns out that you can write your own filters for doing substitutions in files on commit/checkout. These are the "clean" and "smudge" filters. In the .gitattributes file, you can set a filter for particular paths and then set up scripts that will process files just before they’re checked out ("smudge", see Figure 7-2) and just before they’re committed ("clean", see Figure 7-3). These filters can be set to do all sorts of fun things.

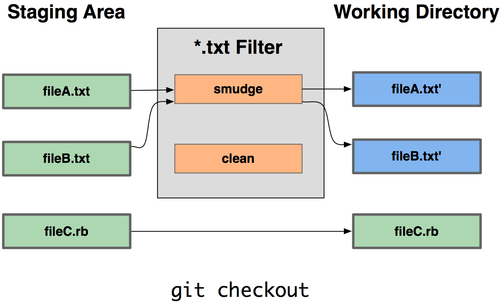


Figure 7-2. The “smudge” filter is run on checkout.

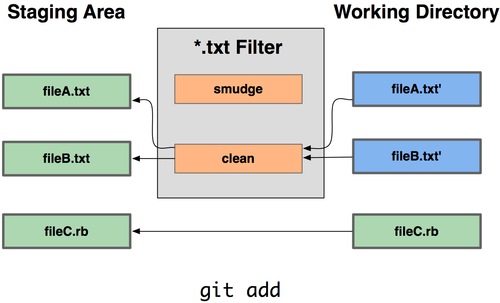


Figure 7-3. The “clean” filter is run when files are staged.

The original commit message for this functionality gives a simple example of running all your C source code through the indent program before committing. You can set it up by setting the filter attribute in your .gitattributes file to filter \*.c files with the "indent" filter:

\*.c filter=indent

Then, tell Git what the "indent" filter does on smudge and clean:

$ git config --global filter.indent.clean indent

$ git config --global filter.indent.smudge cat

In this case, when you commit files that match \*.c, Git will run them through the indent program before it commits them and then run them through the cat program before it checks them back out onto disk. The cat program is basically a no-op: it spits out the same data that it gets in. This combination effectively filters all C source code files through indent before committing.

Another interesting example gets $Date$ keyword expansion, RCS style. To do this properly, you need a small script that takes a filename, figures out the last commit date for this project, and inserts the date into the file. Here is a small Ruby script that does that:

#! /usr/bin/env ruby

data = STDIN.read

last\_date = `git log --pretty=format:"%ad" -1`

puts data.gsub('$Date$', '$Date: ' + last\_date.to\_s + '$')

All the script does is get the latest commit date from the git log command, stick that into any $Date$ strings it sees in stdin, and print the results — it should be simple to do in whatever language you’re most comfortable in. You can name this file expand\_date and put it in your path. Now, you need to set up a filter in Git (call it dater) and tell it to use your expand\_date filter to smudge the files on checkout. You’ll use a Perl expression to clean that up on commit:

$ git config filter.dater.smudge expand\_date

$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]\*\\\$/\\\$Date\\\$/"'

This Perl snippet strips out anything it sees in a $Date$ string, to get back to where you started. Now that your filter is ready, you can test it by setting up a file with your $Date$ keyword and then setting up a Git attribute for that file that engages the new filter:

$ echo '# $Date$' > date\_test.txt

$ echo 'date\*.txt filter=dater' >> .gitattributes

If you commit those changes and check out the file again, you see the keyword properly substituted:

$ git add date\_test.txt .gitattributes

$ git commit -m "Testing date expansion in Git"

$ rm date\_test.txt

$ git checkout date\_test.txt

$ cat date\_test.txt

# $Date: Tue Apr 21 07:26:52 2009 -0700$

You can see how powerful this technique can be for customized applications. You have to be careful, though, because the .gitattributes file is committed and passed around with the project but the driver (in this case, dater) isn’t; so, it won’t work everywhere. When you design these filters, they should be able to fail gracefully and have the project still work properly.

## **Exporting Your Repository**

Git attribute data also allows you to do some interesting things when exporting an archive of your project.

### **export-ignore**

You can tell Git not to export certain files or directories when generating an archive. If there is a subdirectory or file that you don’t want to include in your archive file but that you do want checked into your project, you can determine those files via the export-ignore attribute.

For example, say you have some test files in a test/ subdirectory, and it doesn’t make sense to include them in the tarball export of your project. You can add the following line to your Git attributes file:

test/ export-ignore

Now, when you run git archive to create a tarball of your project, that directory won’t be included in the archive.

### **export-subst**

Another thing you can do for your archives is some simple keyword substitution. Git lets you put the string $Format:$ in any file with any of the --pretty=format formatting shortcodes, many of which you saw in Chapter 2. For instance, if you want to include a file named LAST\_COMMIT in your project, and the last commit date was automatically injected into it when git archive ran, you can set up the file like this:

$ echo 'Last commit date: $Format:%cd$' > LAST\_COMMIT

$ echo "LAST\_COMMIT export-subst" >> .gitattributes

$ git add LAST\_COMMIT .gitattributes

$ git commit -am 'adding LAST\_COMMIT file for archives'

When you run git archive, the contents of that file when people open the archive file will look like this:

$ cat LAST\_COMMIT

Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$

## **Merge Strategies**

You can also use Git attributes to tell Git to use different merge strategies for specific files in your project. One very useful option is to tell Git to not try to merge specific files when they have conflicts, but rather to use your side of the merge over someone else’s.

This is helpful if a branch in your project has diverged or is specialized, but you want to be able to merge changes back in from it, and you want to ignore certain files. Say you have a database settings file called database.xml that is different in two branches, and you want to merge in your other branch without messing up the database file. You can set up an attribute like this:

database.xml merge=ours

If you merge in the other branch, instead of having merge conflicts with the database.xml file, you see something like this:

$ git merge topic

Auto-merging database.xml

Merge made by recursive.

In this case, database.xml stays at whatever version you originally had.

* Git Hooks

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: client side and server side. The client-side hooks are for client operations such as committing and merging. The server-side hooks are for Git server operations such as receiving pushed commits. You can use these hooks for all sorts of reasons, and you’ll learn about a few of them here.

## **Installing a Hook**

The hooks are all stored in the hooks subdirectory of the Git directory. In most projects, that’s .git/hooks. By default, Git populates this directory with a bunch of example scripts, many of which are useful by themselves; but they also document the input values of each script. All the examples are written as shell scripts, with some Perl thrown in, but any properly named executable scripts will work fine — you can write them in Ruby or Python or what have you. These example hook files end with .sample; you’ll need to rename them.

To enable a hook script, put a file in the hooks subdirectory of your Git directory that is named appropriately and is executable. From that point forward, it should be called. I’ll cover most of the major hook filenames here.

## **Client-Side Hooks**

There are a lot of client-side hooks. This section splits them into committing-workflow hooks, e-mail-workflow scripts, and the rest of the client-side scripts.

### **Committing-Workflow Hooks**

The first four hooks have to do with the committing process. The pre-commit hook is run first, before you even type in a commit message. It’s used to inspect the snapshot that’s about to be committed, to see if you’ve forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code. Exiting non-zero from this hook aborts the commit, although you can bypass it with git commit --no-verify. You can do things like check for code style (run lint or something equivalent), check for trailing whitespace (the default hook does exactly that), or check for appropriate documentation on new methods.

The prepare-commit-msg hook is run before the commit message editor is fired up but after the default message is created. It lets you edit the default message before the commit author sees it. This hook takes a few options: the path to the file that holds the commit message so far, the type of commit, and the commit SHA-1 if this is an amended commit. This hook generally isn’t useful for normal commits; rather, it’s good for commits where the default message is auto-generated, such as templated commit messages, merge commits, squashed commits, and amended commits. You may use it in conjunction with a commit template to programmatically insert information.

The commit-msg hook takes one parameter, which again is the path to a temporary file that contains the current commit message. If this script exits non-zero, Git aborts the commit process, so you can use it to validate your project state or commit message before allowing a commit to go through. In the last section of this chapter, I’ll demonstrate using this hook to check that your commit message is conformant to a required pattern.

After the entire commit process is completed, the post-commit hook runs. It doesn’t take any parameters, but you can easily get the last commit by running git log -1 HEAD. Generally, this script is used for notification or something similar.

The committing-workflow client-side scripts can be used in just about any workflow. They’re often used to enforce certain policies, although it’s important to note that these scripts aren’t transferred during a clone. You can enforce policy on the server side to reject pushes of commits that don’t conform to some policy, but it’s entirely up to the developer to use these scripts on the client side. So, these are scripts to help developers, and they must be set up and maintained by them, although they can be overridden or modified by them at any time.

### **E-mail Workflow Hooks**

You can set up three client-side hooks for an e-mail-based workflow. They’re all invoked by the git am command, so if you aren’t using that command in your workflow, you can safely skip to the next section. If you’re taking patches over e-mail prepared by git format-patch, then some of these may be helpful to you.

The first hook that is run is applypatch-msg. It takes a single argument: the name of the temporary file that contains the proposed commit message. Git aborts the patch if this script exits non-zero. You can use this to make sure a commit message is properly formatted or to normalize the message by having the script edit it in place.

The next hook to run when applying patches via git am is pre-applypatch. It takes no arguments and is run after the patch is applied, so you can use it to inspect the snapshot before making the commit. You can run tests or otherwise inspect the working tree with this script. If something is missing or the tests don’t pass, exiting non-zero also aborts the git am script without committing the patch.

The last hook to run during a git am operation is post-applypatch. You can use it to notify a group or the author of the patch you pulled in that you’ve done so. You can’t stop the patching process with this script.

### **Other Client Hooks**

The pre-rebase hook runs before you rebase anything and can halt the process by exiting non-zero. You can use this hook to disallow rebasing any commits that have already been pushed. The example pre-rebase hook that Git installs does this, although it assumes that next is the name of the branch you publish. You’ll likely need to change that to whatever your stable, published branch is.

After you run a successful git checkout, the post-checkout hook runs; you can use it to set up your working directory properly for your project environment. This may mean moving in large binary files that you don’t want source controlled, auto-generating documentation, or something along those lines.

Finally, the post-merge hook runs after a successful merge command. You can use it to restore data in the working tree that Git can’t track, such as permissions data. This hook can likewise validate the presence of files external to Git control that you may want copied in when the working tree changes.

## **Server-Side Hooks**

In addition to the client-side hooks, you can use a couple of important server-side hooks as a system administrator to enforce nearly any kind of policy for your project. These scripts run before and after pushes to the server. The pre hooks can exit non-zero at any time to reject the push as well as print an error message back to the client; you can set up a push policy that’s as complex as you wish.

### **pre-receive and post-receive**

The first script to run when handling a push from a client is pre-receive. It takes a list of references that are being pushed from stdin; if it exits non-zero, none of them are accepted. You can use this hook to do things like make sure none of the updated references are non-fast-forwards; or to check that the user doing the pushing has create, delete, or push access or access to push updates to all the files they’re modifying with the push.

The post-receive hook runs after the entire process is completed and can be used to update other services or notify users. It takes the same stdin data as the pre-receive hook. Examples include e-mailing a list, notifying a continuous integration server, or updating a ticket-tracking system — you can even parse the commit messages to see if any tickets need to be opened, modified, or closed. This script can’t stop the push process, but the client doesn’t disconnect until it has completed; so, be careful when you try to do anything that may take a long time.

### **update**

The update script is very similar to the pre-receive script, except that it’s run once for each branch the pusher is trying to update. If the pusher is trying to push to multiple branches, pre-receive runs only once, whereas update runs once per branch they’re pushing to. Instead of reading from stdin, this script takes three arguments: the name of the reference (branch), the SHA-1 that reference pointed to before the push, and the SHA-1 the user is trying to push. If the update script exits non-zero, only that reference is rejected; other references can still be updated.