Pro Git Reedited

Scott Chacon, edited by Jon Forrest*

2013-11-30

Contents

0.1 Intro to Pro Git Reedited

First of all, I want to first make it clear that I didn't write this book. This is Scott Chacon's book, which he released under a Creative Commons Attribution Non Commercial Share Alike 3.0 license. All I did was edit it. I'm distinguishing Scott's original book from this edited version by calling this version Pro Git Reedited. Naturally, I'm releasing Pro Git Reedited under that same Creative Commons license.

When I started learning Git, I spent a fair amount of time reading Pro Git. I found that it was a 2 steps forward, 1 step back experience. By this I mean I'd learn a couple of new things but then I'd either read something I didn't understand, or else I'd realize that my previous understanding was wrong. But, once I developed a better understanding of Git, I went back to re-read the sections that I didn't previously understand. I'd almost always think to myself that if only this word or that phrase could be changed slightly, the concept would have been much easier to understand. This happens to me a lot when reading technical books.

Given that Scott was generous enough to release Pro Git as a free book with the manuscript sources available at GitHub, I decided to return the favor by doing a complete edit in an attempt to improve the areas I had trouble with and to generally tighten up the text. I've fed all these changes back to the maintainer of Pro Git via GitHub pull requests. He's free to decide what he wants to do with them.

It's crystal clear that Scott knows more about Git than I'll ever know. For this reason, I didn't even attempt to find errors in the text or in the examples. What I did instead was to go over each paragraph, one by one, asking myself if I really understood what it was saying, and whether I could change it into something clearer. As a result, I made a lot of changes. Most of these I'd have a hard time defending because they're very subjective. In fact, it might turn out that I'm overly sensitive and that everybody else is already satisfied with Pro Git. Also, in my efforts to achieve clarity I might have gone too far, and accidentally changed something to be just plain wrong. I'm entirely responsible for any such errors. Please point out any errors and ways to make things even clearer. I intend to keep this book updated with the results of your input.

As I'm writing this intro, I have no idea how Pro Git Reedited will be received. Since I'm no Git expert you won't find any new insights here. Nor is this Pro Git 2.0, which should be a substantially different book than Pro Git, and should contain sections covering the new features added to Git since Pro Git was published. On the other hand, Pro Git explicitly mentions Git changes that were implemented back in Git 1.6. I decided to just merge these into the text since the current release, at the time I'm doing the editing, is 1.8.4.2, and it's doubtful that anybody serious about Git still cares about Git 1.6.

Unless I've made a serious mistake in judgment, I think that Pro Git Reedited can replace Pro Git for online English readers. I'm not sure whether it's worth translating Pro Git Reedited into other languages. In fact, I'd like to think of Pro Git Reedited as simply a collection of English-specific changes to Pro Git that can be ignored in other languages.

I welcome your feedback. Please send any comments to me at `nobozo@gmail.com`. To make sure I recognize them as comments about this book, please include [PGR] in the subject.

The source for this book is at

https://github.com/nobozo/progitreedited

and the PDF version is at
https://www.dropbox.com/s/4awq55350ef235m/progitreedited.en.pdf

Chapter 1

Getting Started

This chapter is about how to get started using Git. It begins with a general introduction to version control systems, moves on to how to get Git running on your system, and finishes with how to start actually using Git. By the end of this chapter you should understand why Git exists and, more importantly, why you should use it.

1.1 About Version Control

What is version control, and why should you care? Version control is a technique for managing changes to files over time. This makes it possible to revert a file back to a previous version, revert an entire project back to a previous version, review changes made over time, see who made a change that might be causing a problem, and more. Even though the examples in this book use version control to manage computer source code files, in reality you can use version control to manage any type of file.

1.1.1 Local Version Control Systems

One popular version control method is to copy files into another directory (perhaps with a name cleverly containing a version number or the current date and time) each time you make a change. This approach is very common because it's so simple, but it's also incredibly error prone. It's easy to forget which directory you should be using and accidentally open the wrong file or copy over files when you don't mean to.

To deal with this issue, programmers long ago developed version control systems (VCSs) based on the concept of a simple version database on the local disk that contains all the changes to their files (see Figure 1-1).

One of the more popular VCSs was a system called RCS, which is still distributed with many computers today. Even the popular Mac OS X operating system includes RCS when you install the Developer Tools. This tool basically works by keeping patch sets (that is, the differences between files) from one revision to another. Using these patch sets, RCS can then recreate what any file looked like at any point in time by applying the necessary patches.

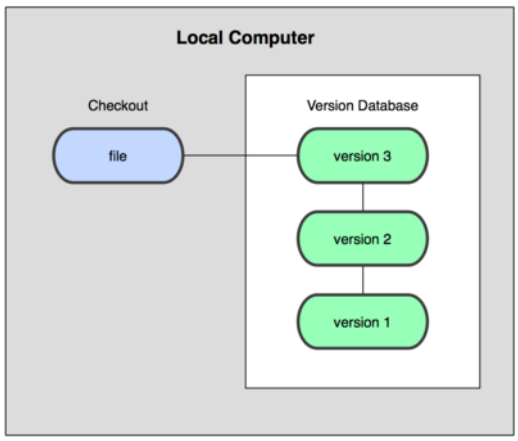

Figure 1.1: Local version control diagram.

1.1.2 Centralized Version Control Systems

The next major issue that people encounter is needing to collaborate with developers on other systems. To deal with this problem, centralized version control systems (CVCSs) were developed. These systems, such as CVS, Subversion, and Perforce, rely on a single server that contains the version database. Clients check files in and out from that remote server. This has been the standard for version control for many years (see Figure 1-2).

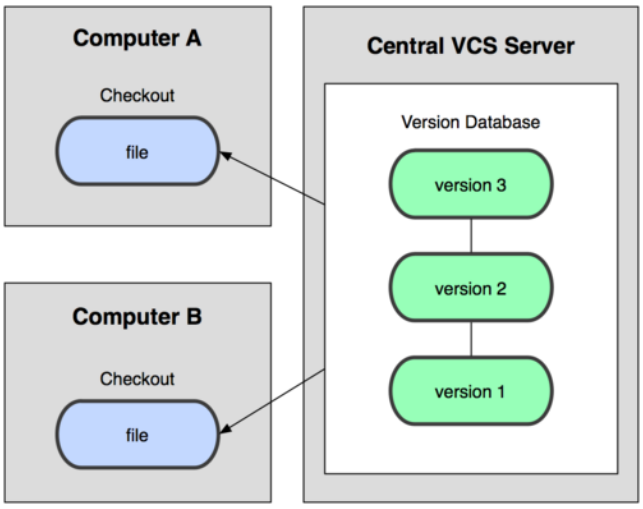

Figure 1.2: Centralized version control diagram.

This approach offers many advantages, especially over local VCSs. For example, everyone can see, to a certain degree, what everyone else working on the project is doing. Administrators can have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local repositories on every client.

However, this approach also has some serious downsides. The most obvious is the single point of failure the centralized server presents. If that server goes down for an hour, then during that time nobody can collaborate at all or save changes to anything they're working on. If the file system the central version database is stored on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything — the entire history of the project except whatever copies people happen to have on their local machines. Local VCS systems suffer from this same problem — whenever you have the entire history of a

project in a single place, you risk losing everything.

The other problem is how to resolve conflicts when multiple developers need to work on the same file at the same time. One extreme solution would be for the first user to lock the file to prevent others from making any changes to it. Problems arise when many developers need to make changes to the same file, or if a developer who locked a file goes on vacation. CVCSs often come with tools to help resolve change conflicts but this work has to be done by hand, and simply isn't acceptable in large projects.

1.1.3 Distributed Version Control Systems

This is where distributed version control systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar, or Darcs), clients don't just check out the latest snapshot of files. Rather, they fully mirror the version database, otherwise known as the repository, on their local disk. Thus, if any server dies, any of the client repositories can be copied back to the server after it's rebuilt. Every client really contains a full backup of the repository (see Figure 1-3).

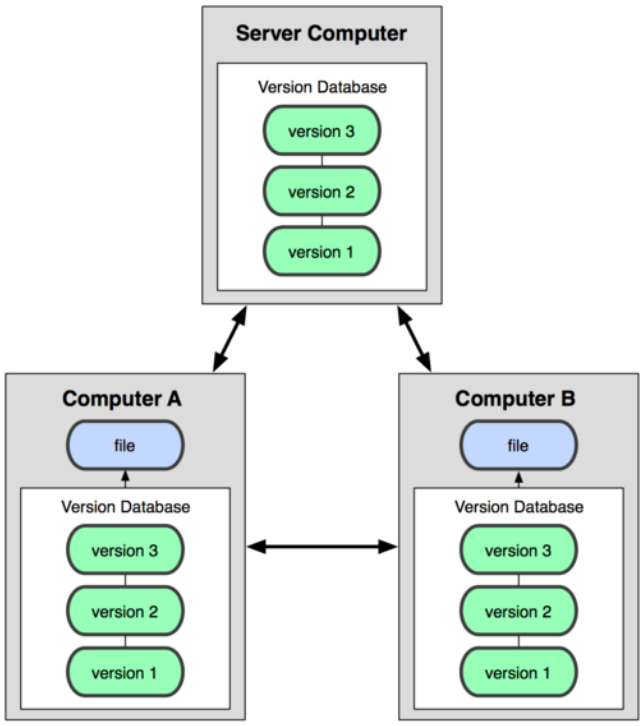

Figure 1.3: Distributed version control diagram.

Furthermore, many of these DVCSs deal pretty well with sharing remote repositories, so you can collaborate with many other people who are all working simultaneously on the same project. This allows doing things in ways that aren't possible in centralized systems.

1.2 A Short History of Git

As with many great things in life, Git began with a bit of creative destruction and fiery controversy. The Linux kernel is an extremely large open source software project — large both in the amount of code and also in the number of people working on it. For most of

the early lifetime of the Linux kernel (1991-2002), changes to its source code were passed around as patches and archived files. This eventually became impossibly unwieldy so in 2002, Linus Torvalds, the creator of Linux, moved the Linux kernel project to a DVCS called BitKeeper. Although BitKeeper was a proprietary product, the company behind it agreed to allow the Linux project to use it free-of-charge, subject to certain conditions.

In 2005, this agreement broke down, and the tool's free-of-charge status was revoked. This prompted the Linux development community to develop their own DVCS based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Since its birth in 2005, Git has evolved and matured, and yet retains these critical qualities. It's incredibly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development (See *Chapter 3*).

1.3 Git Basics

So, what is Git? This is an important question, because if you understand what Git is and the fundamentals of how it works, then using it effectively will be much easier. As you learn Git, try to clear your mind of the things you may know about other VCSs, such as Subversion and Perforce. This will help avoid subtle confusion. Git stores and thinks about information much differently than these other systems, even though the user interface is fairly similar. Understanding those differences is key.

1.3.1 Snapshots, Not Differences

The major difference between Git and other VCSs is the way Git records changes to files. Conceptually, most other systems (CVS, Subversion, Perforce, Bazaar, and so on) organize the information they keep as a set of files along with the changes made to each file over time, as illustrated in Figure 1-4. Also, with some early VCSs you would checkout individual files, make changes to them, and then check them back in again.

Git doesn't work this way. Instead, Git stores its data more like a collection of directory trees. Every time you commit, or save the state of your project, Git basically copies all your files into a new directory tree in the Git repository. This is a bit of an exaggeration — to be efficient, if files haven't changed since the previous commit, Git doesn't store the files again — it just stores a pointer to the previous version already in the repository. Git thinks about its data more like Figure 1-5.

This is an important distinction between Git and nearly all other VCSs. This efficient collection of multiple directory trees allows some incredibly powerful tools to be built. I'll explore some of the benefits Git gains by storing files this way when I cover branching in *Chapter 3*.

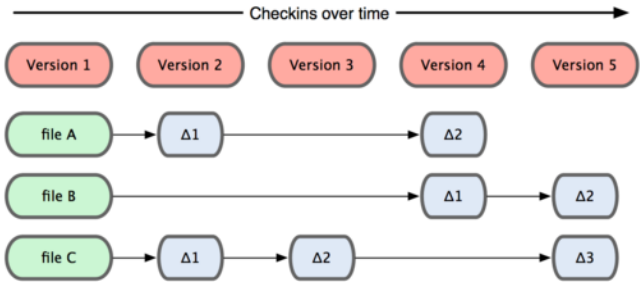

Figure 1.4: Other systems tend to store data as changes to a base version of each file.

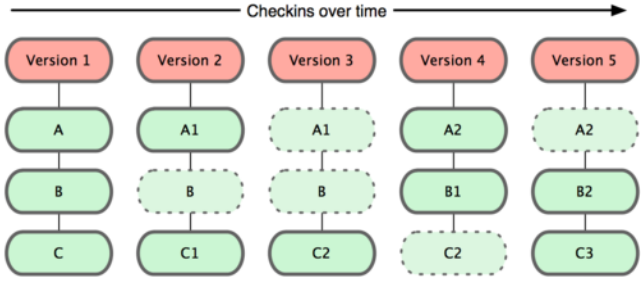

Figure 1.5: Git stores data as snapshots of the project over time.

1.3.2 All Repositories Are Technically Equivalent

Another major difference between Git and other systems is that technically there's no difference between the copies of the repositories located on the workstations of the developers working on the project. The fact that one repository is used as the official project repository is a management decision, not a technical distinction. Sure, it means that all changes must be somehow copied to the official repository, and eventually to the repositories on developer's workstations. Fortunately, as you'll see, Git is very good at doing these things. But the point is that there's no way to recognize that a particular repository is the official project repository simply by looking at it. I'll talk a lot more about this in *Chapter 5*.

1.3.3 Nearly Every Operation Is Local

Most operations in Git only use local resources — generally nothing is needed from another computer on the network. If you're used to a CVCS, where most operations suffer from network latency, this aspect of Git alone will make you think that the gods 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 retrieve the history of a project, Git doesn't need to access a remote server — Git simply reads the history directly from your local repository. This means you see the project history almost instantly. To see the changes between the current version of a file and the version from a month ago, Git can retrieve both versions of the file locally and also compare them on the local machine, instead of having to either ask a remote server to do it or to fetch an older version of the file from a remote server.

This also means that there is very little you can't do when you're offline. If you're on an airplane or a train and want to do a little work, you can do so happily until you get back

online. If you go home and your internet connection is down, you can still work. In many other systems, it's either impossible or painful to get any work done when you're offline. 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 because your repository is inaccessible. This may not seem like a huge deal, but you may be surprised what a big difference it can make.

1.3.4 Git Has Integrity

Every file in Git is checksummed before it's stored. This means it's impossible to change the contents of any file 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 experience file corruption without Git being able to detect it.

The method that Git uses for this checksumming is called an SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0-9 and a-f), and is calculated based on the contents of a file. An SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

You'll see these SHA-1 hash values all over the place in Git because they're used so much. In fact, Git stores everything not by file name but by the SHA-1 hash value of its contents. The only way to reference the file is by that checksum. I'll be talking about this in great detail later.

1.3.5 Git Generally Only Adds Data

When you do something in Git, you almost always only add to the Git repository. It's very difficult to get Git to do anything that is not undoable or that erases data in any way. As in any VCS, you can lose or mess up changes you haven't committed yet. But after you commit a change into Git, it's very difficult to lose.

This makes using Git a joy because I know I can experiment without any danger of screwing things up. For a more in-depth look at how Git stores its data and how to recover data that seems lost, see *Chapter 9*.

1.3.6 The Three Locations

Now it's time to become familiar with the three places that you'll need to be aware of when working with Git. These are the working directory, the staging area, and the Git repository.

The working directory is a directory tree containing a copy of one version of a project. This is where you make modifications to the project. You can put this anywhere on your local disk where you have write permission.

The staging area is something unique to Git. Think of it as a special directory tree that stores a copy of what will go into your next commit. It's sometimes referred to as the index, but it's becoming standard to refer to it as the staging area.

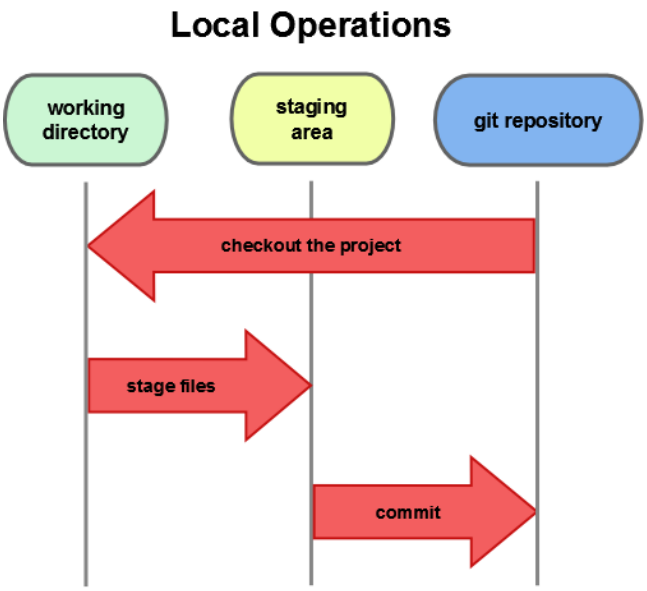

Figure 1.6: Working directory, staging area, and Git repository.

The Git repository is where Git stores everything it needs to keep track of your project. Think of it as holding snapshots of every directory tree you've ever committed. This is the most important part of Git, and it's what's copied when you clone a Git repository from another computer.

1.3.7 The Three States

Now, pay attention. This is the main thing to remember about Git if you want the rest of your learning experience to go smoothly. Git has three states that each file that Git manages can be in: committed, modified, and staged. Committed means that a snapshot containing the file has been safely recorded in the Git repository. Modified means that you've changed the file in the working directory since the last commit. Staged means that you've copied the file into the staging area.

The basic Git workflow goes something like this:

1. Modify files in your working directory.
2. Stage the files, adding copies of them to your staging area.
3. Commit, which creates a snapshot of all the files in the staging area and stores that snapshot permanently in your Git repository.

Files that are either staged or are in the Git repository are also called tracked files. This is another way of saying that Git is managing these files. There's no reason for Git to manage all files in your working directory. After all, some files, like scratch, object, executable, and temporary editor files, can easily be regenerated or have a limited lifespan. You'll learn how to tell Git to ignore files like these. Any files that aren't tracked or ignored are called untracked files.

You'll learn more about these states in *Chapter 2*.

1.4 Installing Git

Let's start using Git. First things first — you have to install it. You can get it a number of ways. The two most common are to install it from source or to use a package manager.

1.4.1 Installing from Source

It's generally best to install the latest version of Git from source. Each new version of Git tends to include useful enhancements and bug fixes, so getting the latest version is often the best route if you feel comfortable building software from source. It's also the case that some Linux distributions contain very old packages so unless you're on a very up-to-date distro, installing from source may be your best bet.

To install Git, you need to have the following libraries that Git depends on: curl, zlib, openssl, expat, and libiconv. For example, if you're on a system that has yum (such as Fedora or RedHat) or apt-get (such as a Debian-based system), use one of these commands to install all of the dependencies.

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

When you've installed all the necessary dependencies, go ahead and grab the latest snapshot from the Git web site.

http://git-scm.com/download

Then, compile and install what you downloaded.

```
$ tar -zxf git-1.8.4.2.tar.gz
$ cd git-1.8.4.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

The version numbers shown here might be different when you read this.

After this is done, you can also update Git using Git itself.

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

1.4.2 Installing on Linux

To install Git on Linux, you can generally use the basic package-management tool that comes with your distribution. If you're on Fedora, use yum.

```
$ yum install git
```

Or, if you're on a Debian-based distribution like Ubuntu, try apt-get.

```
$ apt-get install git
```

1.4.3 Installing on Mac

There are two easy ways to install Git on a Mac. The easiest is to use the graphical Git installer, which you can download from the Google Code page (see Figure 1-7).

http://code.google.com/p/git-osx-installer

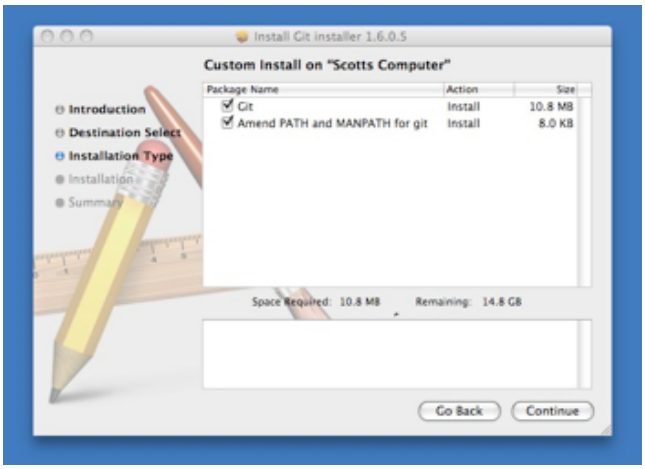

Figure 1.7: Git OS X installer.

The other common way is to install Git via MacPorts (http://www.macports.org). If you have MacPorts installed, install Git via

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

You don't have to add all the extras, but you'll probably want to include +svn in case you ever have to use Git with Subversion repositories (see *Chapter 8*).

1.4.4 Installing on Windows

Installing Git on Windows is very easy. The msysGit project has one of the easier installation procedures. Simply download the installer exe file from the GitHub page, and run it.

http://msysgit.github.com/

After it's installed, you have both a command-line version (including an SSH client that will come in handy later) and a standard GUI version.

A note on Windows usage: you should use Git with the provided msysGit shell (Unix style) since it allows you to use the command line examples shown in this book. If you need, for some reason, to use the native Windows console, you have to use double quotes instead of simple quotes for parameters with spaces in them and you must quote the parameters ending with the circumflex accent (^) if they're last on the command line, since the circumflex accent is a continuation symbol in Windows.

1.5 First-Time Git Setup

Now that you've installed Git, you'll want to do a few things to customize your Git environment. You only have to do these things once. They stick around between upgrades. You can also change them at any time by running the commands again.

The `git config` command gets and sets configuration variables that control many aspects of how Git looks and operates. These variables can be stored in three different files, each of which covers a different level of control.

- `/etc/gitconfig`: contains values used by every user on the system in all repositories. If you add the `--system` option to `git config`, it reads and writes from this file.
- `~/.gitconfig`: contains your personal configuration values for all your repositories. Make Git read and write to this file by adding the `--global` option.
- `.git/config` in whatever Git repository you're currently using: contains configuration values specific to that single repository. This is what's accessed when you run `git config` with no options.

Each level overrides values from the previous level, so values in `.git/config` trump those in `/etc/gitconfig`.

On Windows systems, Git looks for the `.gitconfig` file in the `$HOME` directory (`%USERPROFILE%` in Windows' environment), which is `C:\Documents and Settings\$USER` or `C:\Users\$USER` for most people, depending on Windows version (`$USER` is `%USERNAME%` in Windows' environment). Git also still looks for /etc/gitconfig, although it's relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer.

1.5.1 Your Identity

The first thing you should do after installing Git is to set your user name and e-mail address. This is important because every Git commit uses this information so it's immutably baked into the commits you pass around.

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Again, you only need to do this once if you use the `--global` option because Git always uses that information for anything you do on your system. To override these with a different name or e-mail address for specific projects, run `git config` without the `--global` option when you're in that project's working directory.

1.5.2 Your Text Editor

Now that your identity is set up, configure the text editor that Git uses when you need to enter a message. By default, Git uses your system's default text editor, which is generally Vi or Vim. If you want to use a different text editor, such as Emacs, run

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

1.5.3 Your Diff Tool

Another useful option to configure is the default tool Git uses to resolve merge conflicts. For example, to use vimdiff, run

```
$ git config --global merge.tool vimdiff
```

Git accepts kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, and opendiff. You can also set up a custom tool (see *Chapter 7* for more information about doing that).

1.5.4 Checking Your Settings

To check your settings, run `git config --list` to show all the settings Git can find.

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

You may see keys more than once because Git might read the same key from different files (`/etc/gitconfig` and `~/.gitconfig`, for example). In this case, Git uses the last value for each key it sees.

You can also check the value of a specific key by running `git config {key}`.

```
$ git config user.name
Scott Chacon
```

1.6 Getting Help

If you ever need help while using Git, there are three ways to see manpages for any of the Git commands.

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

For example, you can see the manpage for the `git config` command by running

```
$ git help config
```

If the manpages and this book aren't enough and you need in-person help, try the `#git` or `#github` channel on the Freenode IRC server (irc.freenode.net). These channels are regularly filled with hundreds of people who are all very knowledgeable about Git and are often willing to help.

1.7 Summary

You should now have a basic understanding of what Git is and how it's different from the CVCSs you may have been using. You should also now have a working version of Git on your system that you've set up with your personal identity. It's now time to learn some Git basics.

Chapter 2

Git Basics

If you only read one chapter in this book, this should be the one. This chapter covers every basic command you need to do the vast majority of the things you'll eventually spend your time doing with Git. By the end of the chapter, you should be able to configure and initialize a repository, begin and stop tracking files, and stage and commit changes. I'll also show how to set up Git to ignore certain files and file patterns, how to undo mistakes quickly and easily, how to browse the history of your project and view changes between commits, and how to push and pull from remote repositories.

2.1 Creating a Git Repository

Create a Git repository using either of two methods. The first takes an existing project not managed by Git and puts it under Git control. The second clones an existing Git repository.

2.1.1 Initializing a Repository in an Existing Directory

To start managing an existing project, go to the project's top-level directory and run

```
$ git init
```

This creates a new directory named `.git` that contains all necessary repository files — a Git repository skeleton. At this point, Git isn't managing, or "tracking", anything in your project yet. (See *Chapter 9* for more information about exactly what files are contained in the `.git` directory you just created.)

To start managing existing files, tell Git to manage those files. You can accomplish that with a few `git add` commands that specify the files you want to manage.

```
$ git add *.c
$ git add README
```

Next, put a copy of the managed files into the Git repository by committing the files.

```
$ git commit -m 'initial project version'
```

I'll go over what these commands do in just a minute. Before I do, keep in mind that managing a file and tracking a file mean the same thing. A file is managed, or tracked, when Git is keeping track of the changes to it.

At this point, you have a Git repository containing tracked files and an initial commit.

2.1.2 Cloning an Existing Repository

To get a copy of an existing Git repository — for example, a project you'd like to contribute to — the command is `git clone`. If you're familiar with other VCSs, such as Subversion, you'll notice that the command does a `clone` and not a `checkout`. This is an important distinction — `git clone` receives a copy of nearly everything contained in the repository you're cloning from. Every version of every file for the entire history of the project is transferred when you run `git clone`. In fact, if the disk holding your official project repository gets corrupted, you can use any of the clones of the repository on any client to restore the server back to the state it was in when the clones were done (you may lose some server-side hooks and such, but all the versioned data would be there — see *Chapter 4* for more details).

You clone a repository by running `git clone [url]`. For example, to clone the Ruby Git library called Grit, run

```
$ git clone git://github.com/schacon/grit.git
```

This creates a working directory named `grit`, initializes a `.git` directory inside it, pulls everything from the repository you're cloning from, and checks out a working copy of the latest version into the directory named `grit`. Inside `grit` you'll see the files that make up the project, ready to be worked on. To clone the repository into a working directory named something other than `grit`, specify the directory name you want as a command-line option.

```
$ git clone git://github.com/schacon/grit.git mygrit
```

This command does the same thing as the previous one, but the new working directory is called `mygrit`.

Git supports a number of different transfer protocols. The previous example uses the `git` protocol, but you may also use `http(s)` or `ssh`. *Chapter 4* introduces all of the available options for accessing a remote Git repository, along with their pros and cons.

2.2 Recording Changes to the Repository

You now have a bona fide Git repository and a working directory containing the files in that project. After you've made enough changes to reach a state you want to record, commit a snapshot of your working directory into the repository.

This is an easy process to understand. However, there's an intermediate step that you might find puzzling at first. You don't just directly commit a snapshot from your working directory into the Git repository. Instead, using the `git add` command, first add the files that you want to be part of the next commit into the staging area. Think of the staging area as standing between your working directory and the Git repository. Files in the staging area are called *tracked* files because these are the files that Git is keeping track of.

What about the *untracked* files? It's not hard to imagine that your working directory might contain what I call "throw away" files that are created as part of your development work. Examples of such files are temporary editor files, object files and libraries, and executable files. There's no point in having Git track them because you can always recreate them. You also have no intention of committing them into your Git repository.

When you first clone a repository, all of the files in your working directory will be tracked because you just checked them out from a repository managed by Git. These files are obviously managed by Git.

Files are also *unmodified*, *modified*, or *staged*. An unmodified file hasn't changed since your last commit. As you edit files, Git sees them as modified, because you've changed them since your last commit. You *stage* these modified files then commit all your staged changes, and the cycle repeats. This lifecycle is illustrated in Figure 2-1.

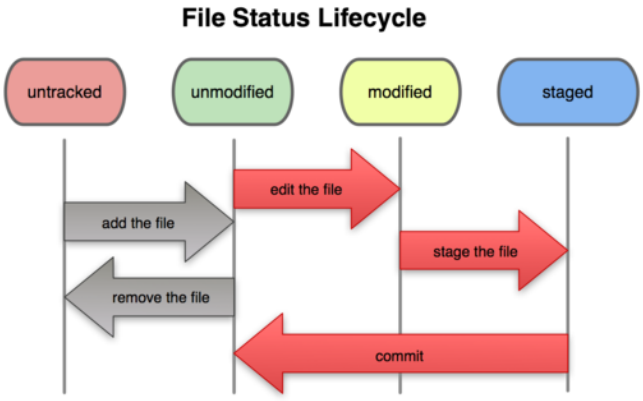

Figure 2.1: The lifecycle of the status of your files.

2.2.1 Checking the Status of Your Files

The command that shows the status of files is `git status`. If you run this command directly after a clone, you see something like

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

This means you have a clean working directory — in other words, all tracked files are unmodified. This means that files in the staging area are identical to the files in the working directory and in the repository. Git also doesn't see any untracked files, or they would be listed here. Finally, the command shows which branch you're on. In this chapter that is always `master`, which is the default. I won't go into branches here but I go over them in detail in the next chapter.

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`, the file appears as untracked.

```
$ vim README
$ git status
# On branch 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 `README` is untracked, because it's in the "Untracked files" section in the `git status` output. Git won't start including it in your commit snapshots until you explicitly tell it to do so. This is so you don't accidentally include throw away files in commits. You do want to start including README, so start tracking it by adding it to the staging area by using the `git add` command, as shown below.

2.2.2 Tracking New Files

To begin tracking a new file, use `git add`. So, to begin tracking the `README` file, run

```
$ git add README
```

If you run `git status` again, `README` appears in a different section in the output.

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   README
#
```

`README` is now under the "Changes to be committed" heading because it's now staged. If you commit at this point, the version of the file at the time you ran `git add` is what will be in the snapshot.

You may recall that when you ran `git init` earlier, you then ran `git add (files)` — that was to begin tracking existing files in your directory. The path name given with the `git add` command can be either for a file or a directory. If it's a directory, the command stages all the files in that directory recursively.

2.2.3 Staging Modified Files

Let's change a file that was already staged. If you change a previously staged file called `benchmarks.rb` and then run `git status` again, you'll see something like

```
$ git status
# On branch 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)
#
# modified:   benchmarks.rb
#
```

The `benchmarks.rb` file appears under a section named "Changes not staged for commit" — which means that a tracked file has been modified in the working directory but the latest version has not yet been staged. To stage it, run `git add` (it's a multipurpose command — use it to begin tracking new files, to stage files, and to do other things that you'll learn about later). Run `git add` now to stage `benchmarks.rb`, and then run `git status` again.

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   README
# modified:   benchmarks.rb
#
```

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 to `benchmarks.rb` before you commit it. You make that change, and you're ready to commit. However, run `git status` one more time.

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   README
# modified:   benchmarks.rb
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
# modified:   benchmarks.rb
#
```

What the heck? Now `benchmarks.rb` is listed as both staged and unstaged. How is that possible? It turns out that Git is seeing two version of `benchmarks.rb`. One is the version that you last staged when you ran `git add`. The other is the current version of `benchmarks.rb` in your working directory. If you commit now, the currently staged version of `benchmarks.rb` is what would go into the commit, not the version in your working directory. Remember, if you modify a file after you run `git add`, you have to run it again to stage the latest version.

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   README
# modified:   benchmarks.rb
#
```

2.2.4 Ignoring Files

Often, there will be a bunch of throw away files that you don't want Git to automatically add or even show as being untracked. These are generally automatically generated files such as log files or files produced by your build system. To make Git ignore such files, create a file named `.gitignore` and put patterns in it that match the filenames you want Git to ignore. Here's an example `.gitignore` file.

```
*.[oa]
```

```
*~
```

The first line tells Git to ignore any files ending in `.o` or `.a` — *object* and *archive* files that may be a byproduct of building your code. The second line tells Git to ignore all files that end with a tilde (~), which is used by many text editors for temporary files. You may also include patterns that match `log`, `tmp`, or `pid` directories, automatically generated documentation, and so on. Setting up a `.gitignore` file 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 that go in a `.gitignore` file are as follows:

- Blank lines or lines starting with `#` are ignored.
- Standard glob patterns work.
- End patterns with a forward slash (`/`) to specify a directory.
- Negate a pattern by starting it with an exclamation point (`!`).

Glob patterns are like simplified shell regular expressions. An asterisk (`*`) matches zero or more characters, `[abc]` matches any character inside the square brackets (in this case `a`, `b`, or `c`), a question mark (`?`) matches any single character, and square brackets enclosing characters separated by a hyphen (`[0-9]`) match any character in a range (in this case 0 through 9).

Here's another example `.gitignore` file.

```
# a comment - this is ignored
# no .a files
*.a
# but do track lib.a, even though you're ignoring .a files above
!lib.a
# only ignore the root TODO file, not subdir/TODO
/TODO
# ignore all files in the build/ directory
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
# ignore all .txt files in the doc/ directory
doc/**/*.txt
```

A `**/` pattern is available in Git since version 1.8.2.

2.2.5 Viewing Your Staged and Unstaged Changes

If the output of `git status` is too vague — you want to know exactly what you changed, not just which files were changed — use the `git diff` command. I'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're about to commit?

Although `git status` answers those questions very generally, `git diff` shows the exact lines added and removed — the patch, as it were.

Let's say you edit and stage `README` again and then edit `benchmarks.rb` without staging it. If you run `git status`, once again you see something like

```
$ git status
# On branch 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)
#
# modified:   benchmarks.rb
#
```

To see what you've changed but not yet staged, run `git diff` with no other arguments.

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           log.size
```

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

To see what you've staged that will go into your next commit, run `git diff --staged`. This command compares what you've staged to your last commit.

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

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. This can be confusing, because if you've staged all of your changes, `git diff` shows nothing.

For another example, if you stage `benchmarks.rb` and then edit it, `git diff` shows both the staged and unstaged changes.

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
# modified:   benchmarks.rb
#
# Changes not staged for commit:
#
# modified:   benchmarks.rb
#
```

Now, run `git diff` to see what's still unstaged.

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
```

```
 main()

 ##pp Grit::GitRuby.cache_client.stats
+# test line
```

and `git diff --cached` to see what you've staged.

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           log.size
```

2.2.6 Committing Your Changes

Now that your staging area contains what you want, commit your changes. Remember that anything that's still unstaged — any files you've created or modified that you haven't run `git add` on since you edited them — won't go into this commit. They will remain as modified files. In this case, 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 run `git commit`.

```
$ git commit
```

This launches your editor of choice. (This is set by your `$EDITOR` environment variable — usually vim or emacs, although you can configure it to be whatever you want using `git config --global core.editor`, as you saw in *Chapter 1*).

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

```
# 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)
#
#       new file:   README
#       modified:   benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

The default commit message contains an empty line on top followed by the commented out output of `git status`. You can remove these comments and type your commit message, or you can leave them in to help you remember what you're committing. (For an even more detailed reminder of what you've modified, add the `-v` option to `git commit`. This also puts the `git diff` output in the editor so you can see exactly what you did). When you exit the editor, Git creates your commit with the commit message you entered but with the comments and diff stripped out.

Alternatively, include your commit message with `git commit` by adding an `-m` flag followed by the message.

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 README
```

You've created your first commit! You can see that the commit resulted in some status output: which branch you committed to (`master`), the SHA-1 hash of the commit (`463dc4f`), how many files were changed, and statistics about the number of lines inserted and deleted in the commit.

Remember that the commit records the snapshot from what's in your staging area. Any changes you didn't stage are still sitting in your working directory. You can stage the files and then do another commit to add them to your repository. Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later.

2.2.7 Skipping the Staging Step

Although the staging area can be amazingly useful for crafting commits exactly how you want them, having to run `git add` can be a bother if you want to stage all the modified files in your working directory. If you want to skip the separate staging step, Git provides a simple shortcut. Adding `-a` to `git commit` automatically stages every modified file before doing the commit, letting you skip the `git add` step.

```
$ git status
# On branch master
#
# Changes not staged for commit:
#
# modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Notice how you didn't have to run `git add benchmarks.rb` before you committed.

2.2.8 Removing Files

To remove a file in your working directory that you haven't staged, you can just remove it using the standard Unix `mv` command. The same is true for ignored files. However, to completely remove a file that you have staged from Git, you have to remove it from the 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 you run `git status`.

If you simply remove the file from your working directory, it shows up in the `git status` output under the "Changes not staged for commit" area.

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#
#       deleted:    grit.gemspec
#
```

Then, when you run `git rm`, Git stages the file's removal.

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
# On branch master
#
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    grit.gemspec
#
```

The next time you commit, the file will be gone and no longer tracked. If you modified and staged the file already, you must force the removal with the `-f` option. This is a safety feature to prevent accidental removal of files that haven't been saved in a snapshot yet, which would prevent them from being recoverable.

Another useful thing to do is keeping the file in your working directory but removing it from your staging area. In other words, you may want Git to stop tracking it. 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` files. To do this, run `git rm --cached`.

```
$ git rm --cached readme.txt
```

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

```
$ git rm log/\*.log
```

Note the backslash (`\`) in front of the `*`. This is necessary because you want Git to do filename expansion instead of your shell. On Windows using the system console, omit the backslash. This command removes all files that have the `.log` extension in the `log/` directory. Or, you can do something like

```
$ git rm \*~
```

which removes all files that end with `~`.

2.2.9 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 showing that you renamed the file. However, Git is pretty smart about figuring that out after the fact — I'll deal with detecting file movement a bit later.

Thus, it's a bit confusing that Git has a `mv` command. To rename a file in Git, run something like

```
$ git mv file_from file_to
```

which works fine. In fact, if you do this and look at the `git status` output, you'll notice that Git sees a renamed file:

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README.txt -> README
#
```

However, this is equivalent to running something like

```
$ mv README.txt README
$ git rm README.txt
$ git add README
```

Git recognizes the rename automatically, so it doesn't matter if you rename a file using `git mv` or with the Unix `mv` command. The only real difference is that `git mv` is one command instead of three so it's more convenient.

2.3 Viewing the Commit History

After you've done several commits, or if you've cloned a repository with an existing commit history, you'll probably want to look back to see what's been committed. The most basic and powerful way to do this is the `git log` command.

These examples use a very simple project called `simplegit` that I often use for demonstrations. To get the project, run

```
git clone git://github.com/schacon/simplegit-progit.git
```

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

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

By default, with no arguments, `git log` lists commits 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 hash, the author's name and e-mail, the date of the commit, and the commit message.

The `git log` command has a huge number and variety of options to express exactly what you're looking for and how to display it. Here are some of the most-used options.

One of the more helpful options is `-p` which shows the changes introduced in each commit. You can also use `-2` along with `-p`, which limits the output to 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

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,5 +5,5 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.name      =   "simplegit"
```

```
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

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
\ No newline at end of file
```

This 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 in a series of commits that a collaborator added.

Sometimes it's easier to review changes by word rather than by line. There's a `--word-diff` option that you can append to the `git log -p` command to see changes this way. Word diff format is quite useless when applied to source code, but it comes in handy when used with large text files, like a book or a dissertation. Here's an example.

```
$ git log -U1 --word-diff
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
```

```
--- a/Rakefile
+++ b/Rakefile
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
    s.name      =   "simplegit"
    s.version   =   [-"0.1.0"-]{+"0.1.1"+}
    s.author    =   "Scott Chacon"
```

As you can see, there are no added and removed lines in this output as in a normal diff. Changes are shown inline instead. The added word is enclosed in `{+ +}` and the removed word enclosed in `[- -]`. You may also want to reduce the usual three line context in diff output to only one line, since the context is now words, not lines. Do this with the `-U1` option, as in the example above.

You can also use a series of summarizing options with `git log`. For example, to see some abbreviated stats for each commit, 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

    changed the 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

    removed unnecessary test code

 lib/simplegit.rb |    5 -----
 1 file changed, 0 insertions(+), 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

 README           |    6 ++++++
 Rakefile         |   23 +++++++++++++++++++++++
 lib/simplegit.rb |   25 +++++++++++++++++++++++++
```

```
 3 files changed, 54 insertions(+), 0 deletions(-)
```

As you can see, the `--stat` option includes a list of modified files below each commit entry, the number of files changed, and the number of lines in those files that were inserted and deleted. It also includes a summary of the information at the end. Another really useful option is `--pretty`. This option changes the log output format to something other than the default. A few prebuilt options are available. The `oneline` option puts 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` options show the output in roughly the same format, but with less or more information, respectively.

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

The most interesting option is `format`, which allows you to specify your own log output format. This is especially useful when you're generating output for a program to parse — 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, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

Table 2-1 lists some of the more useful options that format accepts.

Option	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 e-mail
`%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 patch, whereas the *committer* is the person who last applied the patch. 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. I'll cover this distinction a bit more in *Chapter 5*.

The `oneline` and `format` options are particularly useful with the `--graph` option to `git log`. This adds a nice little ASCII graph showing your branch and merge history, which you can see in your copy of the Grit project repository.

```
$ 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 Added a 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
```

Those are only some simple output-formatting options to `git log` — there are many more. Table 2-2 lists the options I've covered so far and some other common formatting options, along with how they change the output of the `git log` command.

Option	Description
`-p`	Show the patch introduced with each commit.
`--word-diff`	Show the patch in a word diff format.
`--stat`	Show statistics for files modified in each commit.
`--shortstat`	Display only the changed/insertions/deletions line from the –stat command.
`--name-only`	Show the list of files modified after the commit information.
`--name-status`	Show the list of files affected with added/modified/deleted information as well.

`--abbrev-commit`	Show only the first few characters of the SHA-1 checksum instead of all 40.
`--relative-date`	Display the date in a relative format (for example, “2 weeks ago”) instead of using the full dat
`--graph`	Display an ASCII graph of the branch and merge history beside the log output.
`--pretty`	Show commits in an alternate format. Options include oneline, short, full, fuller, and format (w
`--oneline`	A convenience option short for `--pretty=oneline --abbrev-commit`.

2.3.1 Limiting Log Output

In addition to output-formatting options, `git log` takes a number of useful limiting options — that is, options that only show a subset of commits. You’ve seen one such option already — the `-2` option, which shows only the last two commits. In fact, you can use `-<n>`, where `n` is any integer, to show the last `n` commits. In reality, you’re unlikely to need that 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-selection options such as `--since` and `--until` are very useful. For example, this command shows the commits made in the last two weeks.

```
$ git log --since=2.weeks
```

This command accepts lots of different formats, such as a specific date (“2008-01-15”), or a relative date, such as “2 years 1 day 3 minutes ago”.

You can also filter the output to only include commits that match some search criteria. The `--author` option filters on a specific author, and the `--grep` option searches for keywords in commit messages. (Note that to specify both `--author` and `--grep` options, add `--all-match`, otherwise the command will match commits satisfying either option.)

The last really useful option to pass to `git log` as a filter is a path. If you specify a file name, the log output only shows commits that introduced a change to that file. This is always the last option and is generally preceded by double dashes (`--`) to separate the path(s) from the options.

Table 2-3 lists these and a few other common options.

Option	Description
`-(n)`	Show only the last n commits
`--since, --after`	Limit the commits to those made after the specified date.
`--until, --before`	Limit the commits to those made before the specified date.
`--author`	Only show commits in which the author entry matches the specified string.
`--committer`	Only show commits in which the committer entry matches the specified string.

For example, to see which commits modifying test files in the Git source code history

were committed by Junio Hamano in the month of October 2008 and were not merges, run something like

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

Of the nearly 20,000 commits in the Git source code history, this command shows the 6 that match those criteria.

2.3.2 Using a GUI to Visualize History

If you like using a more graphical tool to visualize your commit history, take a look at the Tcl/Tk program `gitk` that's distributed with Git. Gitk is basically a visual `git log` tool, and it accepts nearly the same filtering options that `git log` does. If you run `gitk` in your working directory, you should see something like Figure 2-2.

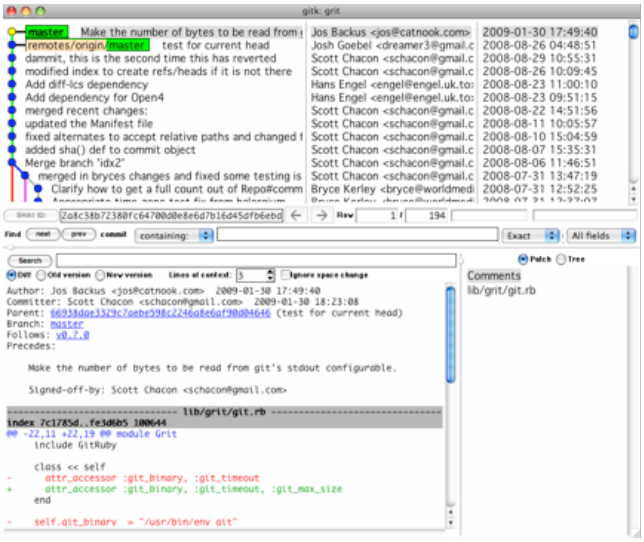

Figure 2.2: The gitk history visualizer.

You can see the commit history in the top half of the window along with a nice ancestry graph. The diff viewer in the bottom half of the window shows the changes introduced in any commit you click.

2.4 Undoing Things

You can change your mind at any point and revert a change that you've already made. I'll review a few basic tools for doing so. Be careful, because you can't always undo these undos. This is one of the few areas in Git where you may lose some work.

2.4.1 Changing Your Last Commit

One of the common reasons for reverting a change is when you commit too early and possibly forget to add some files, or you mess up your commit message. To try that commit again, run

```
$ git commit --amend
```

If you haven't made any changes since your last commit (for instance, you run `git commit --amend` immediately after a commit), then the snapshot in your staging area will look exactly the same as when you made your last commit so all you'll change is your commit message.

Your text editor starts up with the message from your previous commit already in its buffer. The message you create then replaces your previous commit message.

As an example, if you make a commit and then realize you forgot to stage a file you wanted to be in this commit, run

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

After these three commands, you end up with a single commit — the second commit replaces the first.

2.4.2 Unstaging a Staged File

The output from `git status` also describes how to undo changes to the staging area and working directory. 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 *` which stages 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)
```

```
#
#       modified:   README.txt
#       modified:   benchmarks.rb
#
```

Right below the “Changes to be committed” text you see “use `git reset HEAD <file>...` to unstage”. So, follow those directions to unstage `benchmarks.rb`.

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
# 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:   benchmarks.rb
#
```

The output could be clearer, but `git reset` worked. The state of `benchmarks.rb` is back to what it was before you accidentally staged it.

2.4.3 Unmodifying a Modified File

What if you realize that you don’t want to keep your changes to `benchmarks.rb`? How can you easily unmodify it — that is, revert it back to what it looked like when you last committed it? Luckily, `git status` tells you how to do that, too. In the last example, part of the output looks like

```
# 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:   benchmarks.rb
#
```

This shows exactly how to discard the changes you’ve made. Do what it says.

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
```

The changes were reverted. You should also realize that this is a dangerous command: any changes you made to `benchmarks.rb` are gone — you just copied an older version over it. Don't ever use this command unless you're absolutely certain that you don't want the modified file. To just get it out of the way, stashing and branching, covered in the next chapter, are generally better ways to go.

Remember, almost anything you commit in Git can be recovered. Even commits on branches that are deleted or commits that are overwritten with `git commit --amend` can be recovered (see *Chapter 9* for data recovery). However, a lost uncommitted change is gone for good.

2.5 Working with Remotes

To collaborate on a Git project, you need to know how to manage remote repositories. Remote repositories are versions of a project on a host other than your own. You can work on several remote repositories, each of which you could have either read-only or read/write access to. Collaborating with other developers involves configuring your project to access these remote repositories, and pushing and pulling data to and from them when you need to share work. This kind of configuration requires knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches and define them as being tracked or not, and more. In this section, I'll cover these skills.

2.5.1 Showing Your Remotes

Rather than always referring to remote repositories by long URLs, Git lets you define shortnames that you can use in their place. When you clone a Git repository, the shortname `origin` will be defined automatically to refer to the URL from which you cloned the repository. To see which remote repositories you have configured, run `git remote`. It lists the shortnames of each remote repository you've accessed.

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
```

```
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

You can also specify `-v`, which shows the URL that the shortname will be expanded to.

```
$ git remote -v
origin  git://github.com/schacon/ticgit.git (fetch)
origin  git://github.com/schacon/ticgit.git (push)
```

If you have more than one remote, `git remote` lists them all. For example, my Grit repository looks something like

```
$ cd grit
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
cho45     git://github.com/cho45/grit.git
defunkt   git://github.com/defunkt/grit.git
koke      git://github.com/koke/grit.git
origin    git@github.com:mojombo/grit.git
```

This means I can easily pull contributions from any of these repositories. But notice that only `origin` is an SSH URL, so it's the only one I can push to (I'll cover why this is true in *Chapter 4*).

2.5.2 Adding Remote Repositories

In previous sections I showed how cloning automatically adds remote repositories. You can also add remote repositories without cloning them. Here's how. To add a new shortname that refers to a remote Git repository to make it easier to reference the remote, run `git remote add [shortname] [url]`.

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin git://github.com/schacon/ticgit.git
pb git://github.com/paulboone/ticgit.git
```

Now you can use the shortname `pb` on the command line in lieu of the remote's URL. For example, to fetch all the information that Paul has but that you don't yet have in your repository, run `git fetch pb`.

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

You can now access Paul's `master` branch from your repository as `pb/master` — you can merge it into one of your branches, or you can simply look at it. (I'll go over what branches are and how to use them in much more detail in *Chapter 3*.)

2.5.3 Fetching and Pulling from Your Remotes

As you just saw, to get the contents of a remote repository, run

```
$ git fetch [remote-name]
```

Git pulls whatever contents of that remote repository that don't already exist in your local repository. After this, you'll be able to see all the branches from that remote, which you can merge or inspect at any time.

As I said above, when you clone a repository, Git automatically adds that remote repository using the shortname `origin`. So, `git fetch origin` fetches any new work that appears in that repository since you cloned (or last fetched from) it. It's important to note that `git fetch` pulls the data into your local repository — it doesn't automatically merge it with any of your existing work or modify what you're currently working on. Any merging must take place as a separate step.

If you've run `git clone` to create your local repository, run `git pull` to automatically fetch and then merge from the repository you cloned from. This may be an easier or more comfortable way of doing things. This will also be covered in much more detail in *Chapter 3*.

2.5.4 Pushing to Your Remotes

When your project is ready to be shared, push it back to where you originally cloned it from. The command for this is simple: `git push [remote-name] [branch-name]`. To push your `master` branch to your `origin` server, run

```
$ git push origin master
```

Again, running `git clone` created the `origin` and `master` shortnames automatically.

This command works only if you cloned from a repository to which you have write access and if nobody has pushed to the same repository since you made your clone. This is important. If you and someone else clone at the same time and they push to `origin` and then you push to `origin`, your push will be rejected. You'll have to pull down their work first and merge it into your repository before you'll be allowed to push. See *Chapter 3* for more detailed information on how to push to remote servers.

2.5.5 Inspecting a Remote

To see more information about a particular remote, run `git remote show [remote-name]`. For example, If you run this command with the shortname `origin`, you see

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

This lists the URL for the remote repository as well as the tracked remote branch information. The command helpfully tells you that if you're on the `master` branch and you run `git pull`, Git will automatically merge the master branch on the remote into the master branch in your local repository. The output also lists all the remote branches you've pulled already.

2.5.6 Renaming and Removing Remotes

To rename the shortname of a remote repository run `git remote rename`. For instance, if you want to rename `pb` to `paul`, run

```
$ git remote rename pb paul
$ git remote
origin
paul
```

To remove a reference to a remote repository for some reason — perhaps the server no longer exists or a contributor isn't participating anymore — run `git remote rm`.

```
$ git remote rm paul
$ git remote
origin
```

2.6 Tagging

Like most VCSs, Git has the ability to assign tags to specific points in your commit history. Generally, people do this to assign release names (e.g. `v1.0`). In this section, you'll learn how to list tags, create new tags, and what the different types of tags are.

2.6.1 Listing Your Tags

Listing tags is straightforward. Just run `git tag`.

```
$ git tag
v0.1
v1.3
```

This lists the tags in alphabetical order.

You can also search for tags matching a particular pattern. The Git source repo, for instance, contains more than 240 tags. If you're only interested in looking at the 1.4.2 series, run

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

2.6.2 Creating Tags

Git implements 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 almost like a commit in the Git repository. They're checksummed, contain the tagger's name, e-mail, 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 add all this information. But, if you want a temporary tag or for some reason don't need all the information in an annotated tag, lightweight tags are perfectly acceptable.

2.6.3 Lightweight Tags

A lightweight tag is basically a commit SHA-1 hash stored in a file containing nothing else. The name of the file is the tag name. To create a lightweight tag, don't use any options with `git tag`.

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

If you run `git show` on the tag, you see the commit information.

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

2.6.4 Annotated Tags

Another way to tag commits is with an annotated tag, which allows you to include information that doesn't appear in lightweight tags. The first thing is to include a message about the commit in the tag. Specify the `-a` and `-m` options to create an annotated tag.

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

The `-a` specifies the tag name and the `-m` specifies a message. Both are stored with the tag. If you don't specify a message for an annotated tag, Git launches your editor so you can enter the message.

You can see the tag data along with the corresponding commit information by running the `git show` command.

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

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

2.6.5 Signed Tags

You can also sign your tags with GPG, assuming you have a private signing key. All you have to do is use `-s` instead of `-a`.

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

If you run `git show` on that tag, you see your GPG signature attached to it.

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/
=WryJ
-----END PGP SIGNATURE-----
```

```
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

2.6.6 Verifying Tags

To verify a signed tag, run `git tag -v [tag-name]`. This uses GPG to verify the signature. You need the signer's public key in your keyring for this to work properly.

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

If you don't have the signer's public key, you see something like this instead.

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

2.6.7 Tagging Later

You can also tag commits you made in the past. Suppose your commit history looks like

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
```

```
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Now, suppose you forgot to assign a `v1.2` tag, which should point to the "updated rakefile" commit. You can add it after the fact. To tag that commit, specify the commit SHA-1 hash (or part of it) at the end of the `git tag` command.

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```

You can see that you created the tag and the commit you tagged.

```
$ 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

    updated rakefile
...
```

2.6.8 Sharing Tags

By default, `git push` doesn't transfer tags to remote servers. If you do want to transfer tags you have to do this explicitly, which is just like sharing remote branches — run `git push origin [tagname]`.

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]         v1.5 -> v1.5
```

If you have a lot of tags to push at once, use the `--tags` option to `git push`. This transfers all your tags that aren't already on the remote server.

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
 * [new tag]         v0.1 -> v0.1
 * [new tag]         v1.2 -> v1.2
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
 * [new tag]         v1.5 -> v1.5
```

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

2.7 Tips and Tricks

Before I finish this chapter on basic Git, I want to mention a few little tips and tricks that may make your Git experience a bit more pleasant. Many people use Git without using any of these tips, and I won't refer to them later in the book, but you should know about them.

2.7.1 Auto-Completion

If you use the Bash shell, Git comes with a nice auto-completion script you can enable. Download the Git source code, and look in the `contrib/completion` directory. There should

be a file there called `git-completion.bash`. Copy this file to your home directory, and add this to your `.bashrc` file.

```
source ~/.git-completion.bash
```

To set up Git so that all users automatically use this script, copy it to the `/opt/local/etc/bash_completion.d` directory on Mac systems or to the `/etc/bash_completion.d/` directory on Linux systems. This is a directory of scripts that Bash automatically loads to provide auto-completion.

If you're using Windows with Git Bash, which is the default when installing Git on Windows with msysGit, auto-completion is preconfigured.

Press the Tab key when you're entering a Git command, and it should display a set of suggestions for you to pick from.

```
$ git co<tab><tab>
commit config
```

In this case, typing `git co` and then pressing the Tab key twice suggests commit and config. Adding `m<tab>` selects `git commit` automatically.

This also works with command options, which is probably more useful. For instance, if you're trying to run `git log` and can't remember one of the options, start typing the command and then press the Tab key to see what matches.

```
$ git log --s<tab>
--shortstat  --since=  --src-prefix=  --stat   --summary
```

That's a pretty nice trick and may save you some time.

2.7.2 Git Aliases

Git doesn't guess a command if you only partially enter it. If you don't want to type the entire text of each of the Git commands you commonly use, you can easily set up an alias for the commands by running `git config`. Here are a couple of examples.

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

So, for example, instead of typing `git commit`, just type `git ci`. As you continue using Git, you'll probably use other commands frequently as well so don't hesitate to create new aliases.

This technique can also be very useful in creating commands that you think should exist but don't. For example, to correct the usability problem you encountered with unstaging a file, add your own unstage alias.

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

This makes the following two commands equivalent:

```
$ git unstage fileA
$ git reset HEAD fileA
```

The first seems a bit clearer. It's also common to add a `git last` command, like this.

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

This way, you can easily see the last commit.

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

As you can tell, Git simply replaces what you type with the alias. However, maybe you want to run a Linux command rather than a Git command. In that case, start the alias string with a `!` character. This is useful if you write your own tools that work with Git. I alias `git visual` to run `gitk`.

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

2.8 Summary

At this point, you know how to do all the basic local Git operations — creating or cloning a repository, making changes, staging and committing those changes, and viewing the history of all the changes contained in a repository. Next, we'll cover Git's killer feature: its branching model.

Chapter 3

Git Branching

Nearly every VCS has some form of support for branching. Branching is when you diverge from one line of development and start working on another line. In many VCS tools, this is a somewhat expensive process, often requiring creating a new copy of your source code directory, which can take a long time and use up a lot of disk space for large projects.

Some people refer to Git's branching model as its "killer feature", and it certainly sets Git apart in the VCS community. Why is it so special? Git branches are incredibly lightweight, making branching operations nearly instantaneous. Unlike many other VCSs, Git encourages frequent branching and merging, even multiple times a day. Understanding and mastering branching gives you a powerful and unique tool and can literally change the way that you develop.

3.1 What a Branch Is

To really understand the way Git does branching, step back and examine how Git stores its data. As you may remember from Chapter 1, Git doesn't store a series of changesets or diffs, but instead a series of snapshots. Each snapshot is represented in the Git repository by a commit object. Each commit object contains, among other things, zero or more pointers to the commits that precede it. The first commit object in a repository doesn't contain any pointers — since it's the first commit there's nothing to point to. A normal commit contains just one pointer, which points to the commit that comes before it. There's a special kind of commit, called a merge, that contains multiple pointers because it merges two or more branches together.

In addition to these pointers, a commit object also stores information about the author and committer of the commit, a message summarizing the commit, and the size of the commit object itself. Finally, Git computes the SHA-1 hash of the commit object. Since this hash is guaranteed to be different for each commit object (can you see why?) the hash serves as a unique identifier for each commit. For the rest of this chapter you only need to pay attention to commit objects since they're all that's necessary for understanding branching. I'll go into more details about the other kinds of objects in Chapter 9.

To visualize this, assume that you have a directory containing three files which you stage and commit into an empty Git repository.

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

When you run `git commit`, Git creates a commit object with an SHA-1 hash that starts with 98ca9. Conceptually, your Git repository looks something like Figure 3-1.

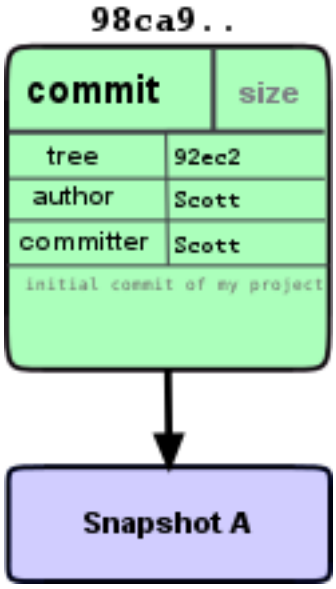

Figure 3.1: Single commit repository data.

If you make some changes then stage and commit again, the new commit object stores a pointer to the preceding commit object. After two more commits, your repository might look something like Figure 3-2.

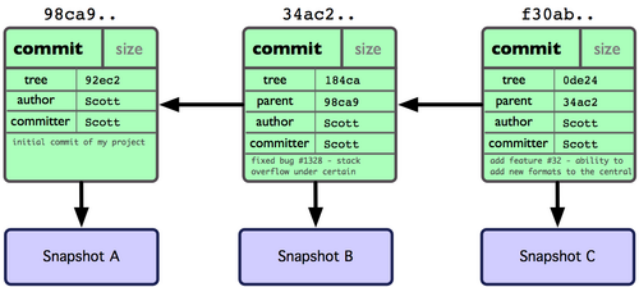

Figure 3.2: Git object data for multiple commits.

A branch in Git is simply a named pointer to a commit. The default branch name in Git is `master`. As you make commits, Git automatically moves `master` to point to the last commit you made, as shown in Figure 3-3.

A repository with just one branch isn't very interesting. What happens when you create a new branch? This simply creates a new named pointer — nothing more, nothing less. To illustrate, let's say you create a new branch called `testing` by running `git branch`.

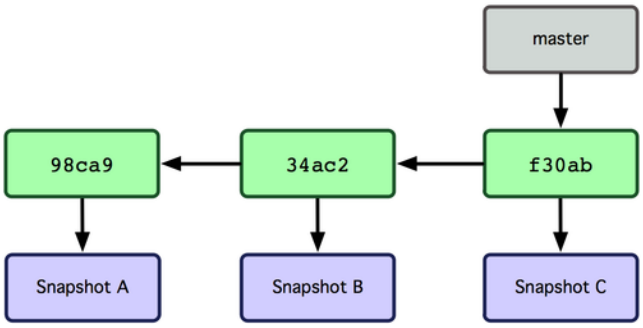

Figure 3.3: Branch pointing into the commit history.

```
$ git branch testing
```

This creates a new pointer called `testing`. It initially points to the same commit as the branch you're currently on (see Figure 3-4).

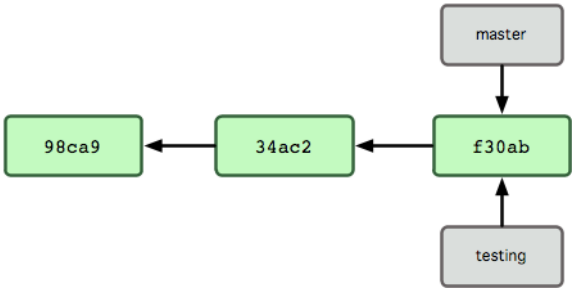

Figure 3.4: Multiple branches pointing into the commit's data history.

How does Git know what branch you're currently on? Git maintains a special pointer, called `HEAD`, that always points to the branch you're currently on. 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. The `git branch` command only creates a new branch — it doesn't switch to that branch by changing what `HEAD` points to (see Figure 3-5). So, at this point, you're still on `master`.

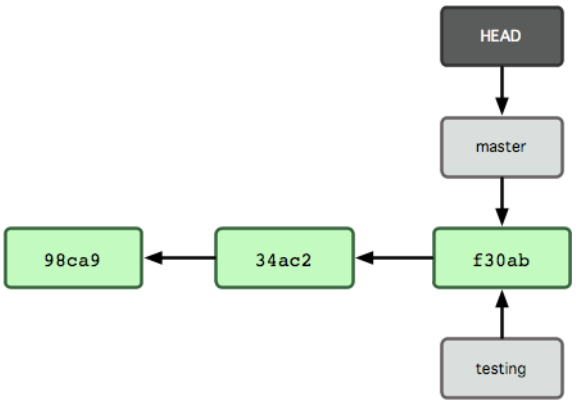

Figure 3.5: HEAD file pointing to the branch you're on.

To switch to a different branch, run `git checkout`. Let's switch to the new `testing`

branch.

```
$ git checkout testing
```

This changes HEAD to point to the `testing` branch (see Figure 3-6).

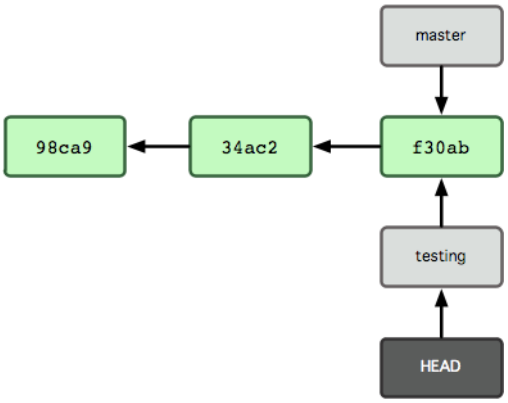

Figure 3.6: HEAD points to a different branch when you switch branches.

That's nice but now there are three pointers to the same commit `f30ab`. What's the point of this? Well, let's do another commit.

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

Figure 3-7 illustrates the result.

This is interesting, because now your `testing` branch pointer has moved forward to point to the new commit, but your `master` branch pointer didn't change. Let's switch back to the `master` branch.

```
$ git checkout master
```

Figure 3-8 shows the result.

That did two things. It moved the HEAD pointer back to point to the `master` branch, and it quickly reverted the files in your working directory back to the snapshot that `master` points to. This ability to quickly revert back to a previous snapshot is something that seems almost miraculous at first. In other VCSs, switching between branches can take a noticeable amount of time. In Git, it's almost instantaneous. I show how this is done so quickly in later chapters.

Changes you make from this point forward will diverge from what's on the `testing` branch. Git essentially rewinds the work you've done on your testing branch temporarily so you can go in a different direction.

Let's make a few changes and commit again.

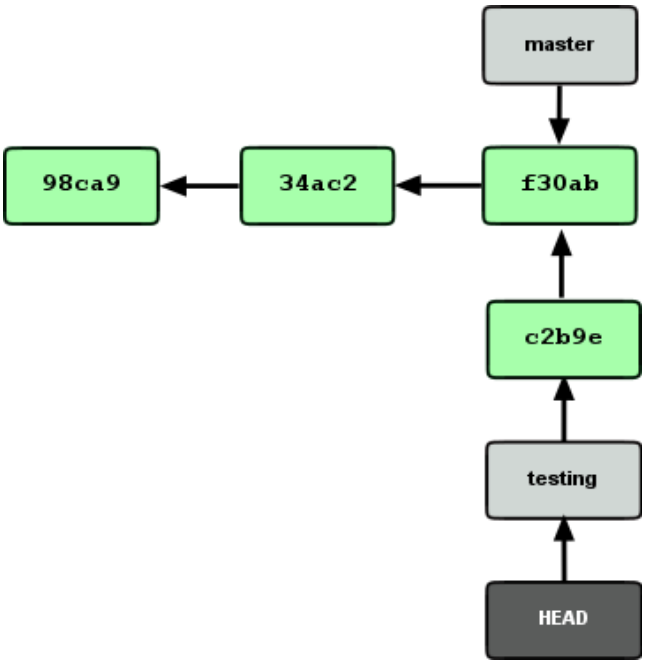

Figure 3.7: The branch that HEAD points to moves forward with each commit.

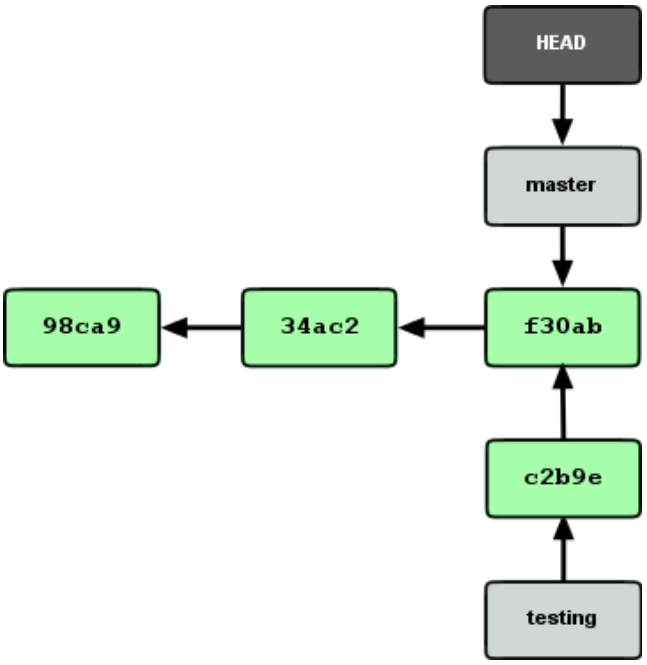

Figure 3.8: HEAD moves to another branch on a checkout.

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Now your project history has diverged (see Figure 3-9). You created and switched to a branch, did some work on it, and then switched back to your other branch and did other work. All this work is isolated in separate branches: you can switch back and forth between the branches and merge them together when you're ready. You did all that with simple `git`

`branch` and `git checkout` commands.

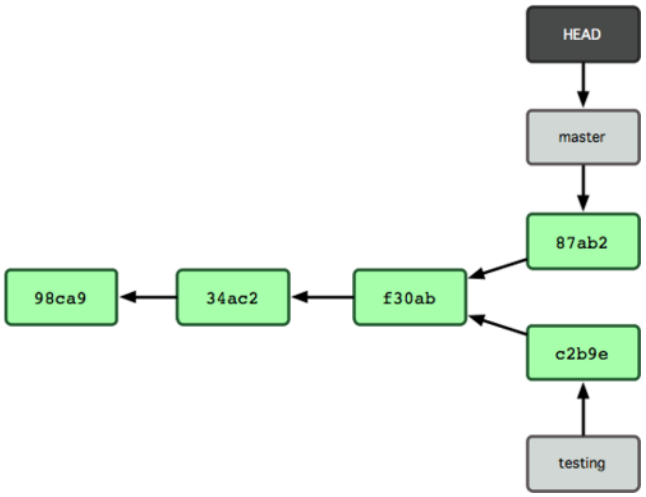

Figure 3.9: The branch histories have diverged.

Because a branch in Git is actually a simple file that contains the 40 character SHA-1 hash of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline). Also, because Git records the parent commit(s) when you commit, it's easy for Git to find the snapshots it needs to do merging.

Let's see how to do some branching and merging.

3.2 Basic Branching and Merging

Let's go through a simple example of branching and merging that might be similar to what you'd do in the real world. Follow these steps:

1. Do work on the master branch for your company's web site.
2. Create a branch for a new story you're working on.
3. Do some work in the new story branch.

At this stage, you receive a call that the web site has a critical issue and you need to develop a hotfix. You do the following:

1. Revert back to the master branch.
2. Create a branch to add the hotfix.
3. After it's tested, merge the hotfix branch into the master branch, and put the new code into production.
4. Switch back to your new story branch and continue working.

3.2.1 Basic Branching

First, let's say you're working on a project and have a couple of commits already in the master branch (see Figure 3-10).

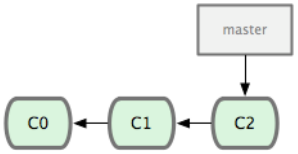

Figure 3.10: A short and simple commit history.

You've decided that you're going to work on issue #53 in the issue-tracking system your company uses. To do this, create a new branch to work in. To create the branch `iss53` and switch to it at the same time, run `git checkout` 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 3-11 illustrates the result.

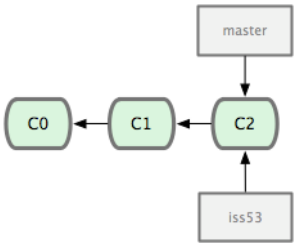

Figure 3.11: Creating a new branch pointer.

Work on your web site and do some commits. Doing so moves the `iss53` branch forward since this is your current branch, as shown in Figure 3-12.

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

Now you get a call that there's an issue with the web site, and you need to fix it immediately. At first you're worried because the changes in the `iss53` branch aren't finished yet. With Git, you don't have to deploy your fix along with your `iss53` changes, and you don't have to revert those changes before you can work on your fix. 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 content on the branch you're checking out, Git won't let you switch branches. You'll see an error message like the following:

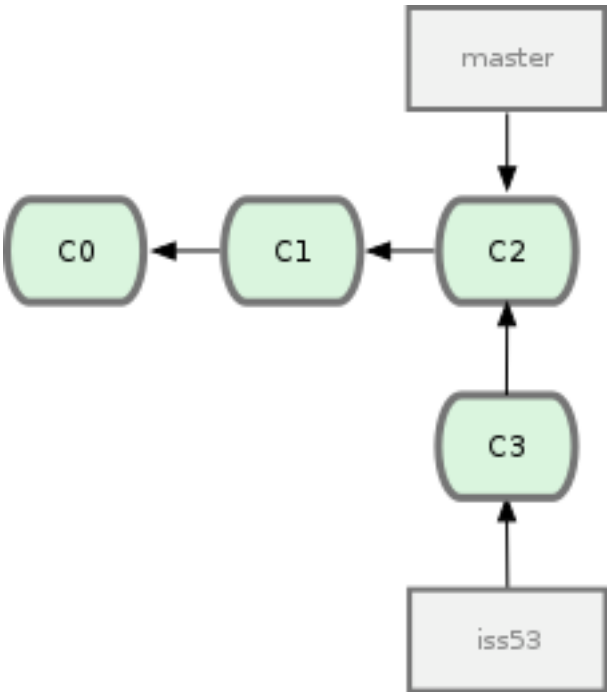

Figure 3.12: The iss53 branch has moved forward with your work.

```
$ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
```

followed by a list of the files containing uncommitted changes.

It's best to have a clean working state when you switch branches. There are ways to get around this that I'll cover later. For now, 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 working directory is exactly the way it was before you started working on issue #53 so you can concentrate on your hotfix. This is an important point to remember: Git restores your working directory from the snapshot of the commit pointed to by the branch you check out. It adds, removes, and modifies files automatically to make this happen.

Next, you have a hotfix to make. Create a `hotfix` branch to contain the fix, create a fix, and then commit it (see Figure 3-13).

```
$ git checkout -b hotfix
Switched to a new branch "hotfix"
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
```

```
[hotfix]: created 3a0874c: "fixed the broken email address"
 1 file changed, 0 insertions(+), 1 deletion(-)
```

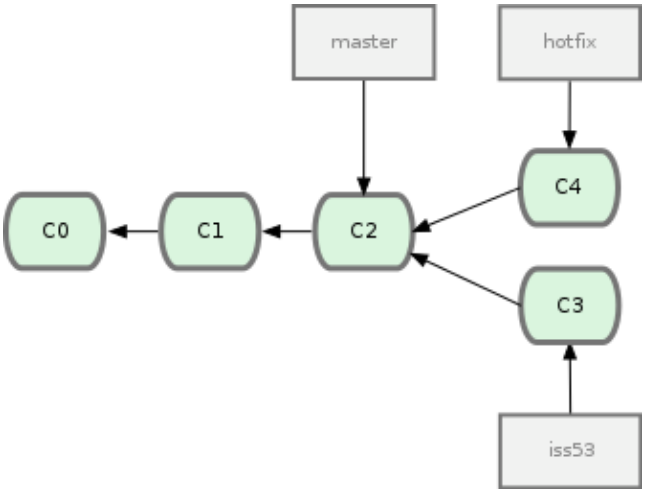

Figure 3.13: hotfix branch based back at your master branch point.

Run your tests to make sure the hotfix fixes the bug, and merge it back into your `master` branch, which will then be ready to deploy to production. Do this with the `git merge` command.

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README |    1 -
 1 file changed, 0 insertions(+), 1 deletion(-)
```

Notice the phrase “Fast forward” in that output. Because the commit pointed to by the branch you merged in (C4) was directly upstream of the commit you’re on (C2), Git simply moves the `master` branch pointer forward (to C4). To phrase that another way, when you try to merge one commit (C4) with a commit (C2) that can be reached by following the first (C4) commit’s history, Git simplifies things by moving the pointer forward because there’s 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 your change (see Figure 3-14).

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 delete the `hotfix` branch, because you no longer need it — the `master` branch points to the same place. Delete it by running `git branch -d`.

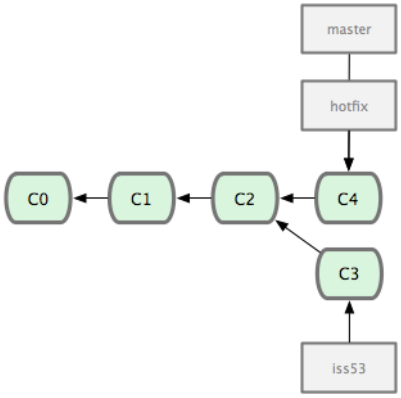

Figure 3.14: Your master branch points to the same place as your hotfix branch after the merge.

```
$ git branch -d hotfix
Deleted branch hotfix (was 3a0874c).
```

Now switch back to `iss53`, your work-in-progress branch and continue working on it. Commit when you're finished. (see Figure 3-15).

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
 1 file changed, 1 insertion(+), 0 deletions(-)
```

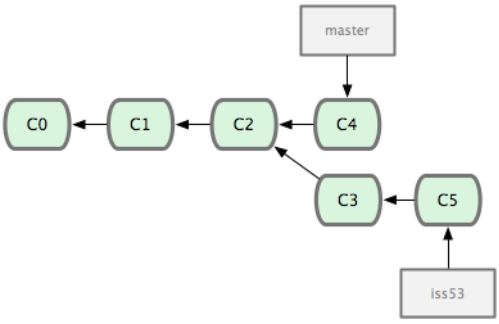

Figure 3.15: Your iss53 branch can move forward independently.

It's worth noting here that the work you did on what was your `hotfix` branch (now in C4) is not contained in the commits on your `iss53` branch (C3 and C5). If you need to include that work on your `iss53` branch, merge your `master` branch into your `iss53` branch now by running `git merge master`, or wait to integrate those changes until you decide to merge the `iss53` branch back into `master` later.

3.2.2 Basic Merging

Suppose you've decided that your `iss53` work is complete and ready to be merged into your `master` branch. In order to do that, merge in your `iss53` branch, much like you merged in your `hotfix` branch earlier. All you have to do is check out the branch you wish to merge into and then run `git merge`.

```
$ git checkout master
$ git merge iss53
Merge made by recursive.
 README |    1 +
 1 file changed, 1 insertion(+), 0 deletions(-)
```

This looks different than the `hotfix` merge you did earlier. In this case, your development history has diverged from some older common ancestor (C2). Because the commit (C4) on the branch you're on isn't a direct ancestor of the latest commit (C5) in the branch you're merging in, Git has work to do. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips (C4 and C5) and their common ancestor (C2). Figure 3-16 highlights the three snapshots that Git uses to do its merge in this case.

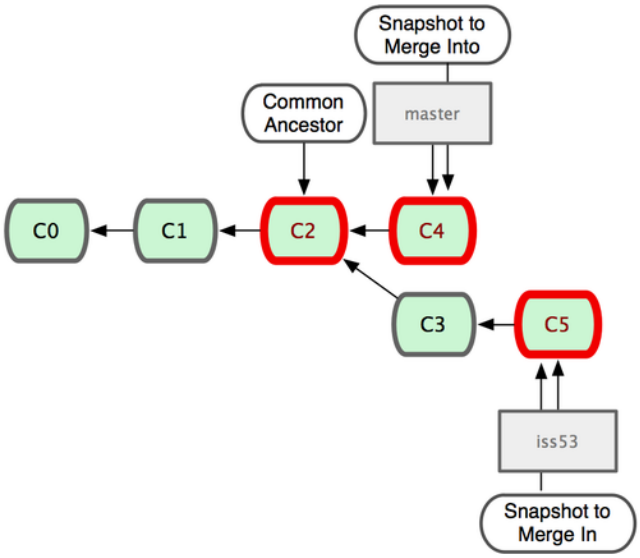

Figure 3.16: Git automatically identifies the best common-ancestor merge base for branch merging.

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 object (C6) that points to it (see Figure 3-17). This is referred to as a merge commit and is special in that it has more than one parent.

It's worth pointing out that Git determines the best common ancestor to use for its merge base. This is different than CVS or Subversion (before version 1.5), where the developer doing the merge has to figure out the best merge base. This makes merging a heck of a lot easier in Git than in these other systems.

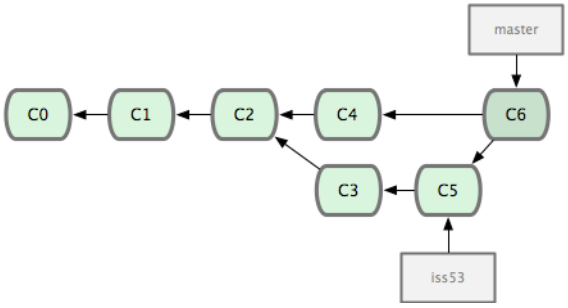

Figure 3.17: Git automatically creates a new commit object that contains the merged work.

Now that your work is merged in, you have no further need for the `iss53` branch. Delete it and then manually close the ticket in your ticket-tracking system:

```
$ git branch -d iss53
```

3.2.3 Basic Merge Conflicts

Occasionally, merging doesn't go smoothly. If you changed the same part of the same file in different ways in the two branches you're merging, Git won't be able to merge them automatically. For example, if your fix for issue #53 modified the same part of `index.html` that you modified in the `hotfix` branch, you'll get a merge conflict that looks something like

```
$ 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 didn't automatically create a new merge commit. Instead, you have to first manually resolve the conflict. To see which files are unmerged at any point after a merge conflict, run `git status`.

```
[master*]$ git status
index.html: needs merge
# On branch 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)
#
# unmerged:   index.html
#
```

Anything that has merge conflicts that haven't been resolved appears as unmerged. Git adds standard conflict-resolution markers in the files that have conflicts, so you can open them and manually find and resolve those conflicts. In this case, `index.html` contains a section that looks something like

```
<<<<<<< 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 pointed to by HEAD (your `master` branch, because that was your current branch when you ran the merge command) is in the top part of that output (everything above the `=======`), while the version in your `iss53` branch is in the bottom part. In order to resolve the conflict, either choose one version or the other, or merge the contents yourself. For instance, you might resolve this conflict by removing the the entire block marked by the `<<<<<<<` and `>>>>>>>` lines in `index.html` on your `master` branch and replacing it with this.

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

This resolution contains a little from each possibility. After you've resolved a conflicted file, run `git add` on it. Staging the file marks the merge conflict as resolved. To use a graphical tool to resolve these issues, run `git mergetool`, which fires up an appropriate visual merge tool and walks you through the conflicts.

```
$ git mergetool
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
Merging the files: index.html

Normal merge conflict for 'index.html':
  {local}: modified
  {remote}: modified
Hit return to start merge resolution tool (opendiff):
```

To use a merge tool other than the default, enter the name of the tool you'd rather use from the "merge tool candidates" list. In Chapter 7, I'll discuss how to change this default.

After you exit the merge tool, Git asks if the merge was successful. If you say yes, Git stages the file to mark it as resolved.

Run `git status` again to verify that all conflicts have been resolved.

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   index.html
#
```

If everything that had conflicts has been staged, run `git commit` to finalize the merge commit. The commit message by default looks something like

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
#
```

Modify that message with details about how you resolved the merge if you think it would be helpful to others looking at this merge in the future — why you did what you did, if it's not obvious.

3.3 Branch Management

Now that you've created, merged, and deleted some branches, let's look at some branch management techniques 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, it shows a list of the branches in your repository.

```
$ git branch
  iss53
* master
  testing
```

Notice the `*` character in front of `master`: it indicates your current branch. This means that if you commit now, the commit will go on the `master` branch. The `master` pointer will be moved forward to point to the new commit. To see the last commit on each branch, 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 readmes
```

Another useful option for examining the state of your branches is to filter the output from `git branch -v` to show which branches need to be merged into your current branch, and which branches have already been merged. The `--merged` and `--no-merged` options to `git branch` do this. To see which branches are already merged into your current branch, run `git branch --merged`.

```
$ git branch --merged
  iss53
* master
```

Because you already merged in `iss53` earlier, you see it in the output. Branches without the `*` in front are generally safe to delete. Since you've already merged their contents into another branch, there's no reason not to delete them.

To see all the branches that contain work you haven't yet merged in, run `git branch --no-merged`.

```
$ git branch --no-merged
  testing
```

This shows any branches containing work that isn't merged in yet. Trying to delete such a branch with `git branch -d` will fail.

```
$ git branch -d testing
error: The branch 'testing' is not an ancestor of your current HEAD.
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 the work it contains, force the deletion with `-D`, as the helpful message points out.

3.4 Branching Workflows

Now that you have the basics of branching and merging down, I'll cover some common workflows that Git's lightweight branching makes possible, so you can decide whether to incorporate them into the way you work.

3.4.1 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 always open to use for different stages of your development cycle. Merge them early and often.

Many Git developers have a workflow that embraces this approach, having only entirely stable code in their `master` branch — possibly only code that has been or will be released. They have another parallel branch named `develop` 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, which are short-lived branches, like your earlier `iss53` branch, when they're ready. This approach requires topic branches pass all tests before going into `master`.

In reality, this approach simply moves branch pointers along the line of commits you're making. The stable branches are farther to the left in your commit history, and the bleeding-edge branches are farther to the right (see Figure 3-18).

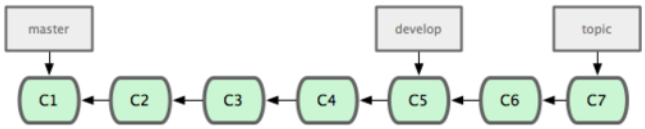

Figure 3.18: More stable branches are generally farther to the left in commit history.

It's generally easier to think about a collection of silos, where sets of commits graduate to a more stable silo when they're fully tested (see Figure 3-19).

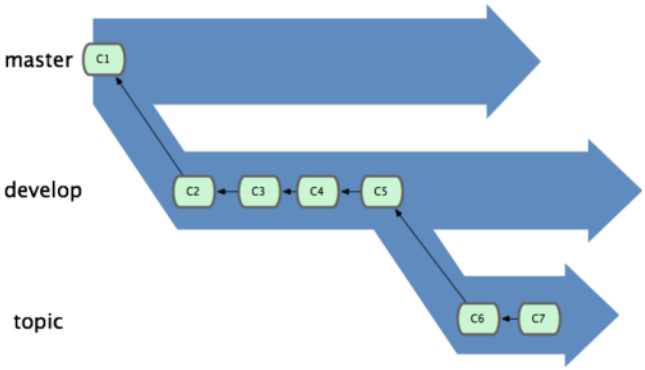

Figure 3.19: It may be helpful to think of your branches as silos.

Keep doing this for multiple stability levels. Some larger projects also have a `proposed` branch containing 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, merge them into the branch above. Multiple long-running branches aren't necessary, but they're often helpful, especially when dealing with very large or complex projects.

3.4.2 Topic Branches

Topic branches, however, are useful in projects of any size. Again, a topic branch is a short-lived branch for a single particular purpose. 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 do so several times a day.

You saw this in the last section with the `iss53` and `hotfix` branches. You did a few commits in them and deleted them 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 have to do with a specific topic, it's easier to see how the code changes over time. You can keep the changes in the topic branches for minutes, days, or months, and merge them in when they're ready, regardless of the order in which they were created or modified.

Consider an example of doing some work on `master` (C0 and C1), creating the `iss91` branch off of `master` for an issue, working on `iss91` for a bit (C2 and C3), creating the `iss91v2` branch off of `iss91` to try another way of handling the same issue (C4, C5, and C6), going back to `iss91` to try a few more clever ideas (C7 and C8), going back to `master` and working there for a while (C9, C10, and C11), and then creating `dumbidea` off of `master` to do some work that you're not sure is a good idea (C12 and C13). Your commit history will look something like Figure 3-20.

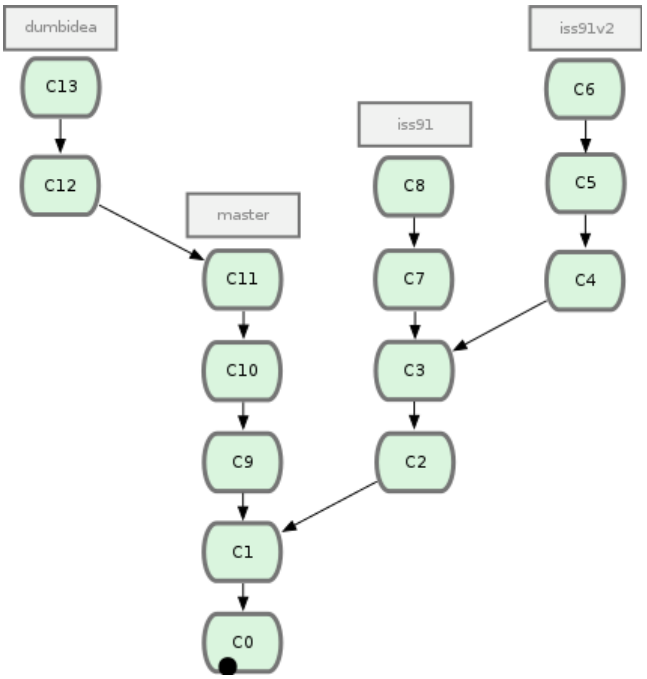

Figure 3.20: Your commit history with multiple topic branches.

Now, let's say you decide you like the second solution to your issue best (`iss91v2`). You showed the `dumbidea` branch to your coworkers, and they thought it was pure genius. Throw away the original `iss91` branch (losing commits C7 and C8) and merge in the other two (C2 and C3). Finally, merge `dumbidea` (C12 and C13) and `iss91v2` (C4, C5, and C6)

onto `master` to be your new production version (C14). Your history then looks like Figure 3-21.

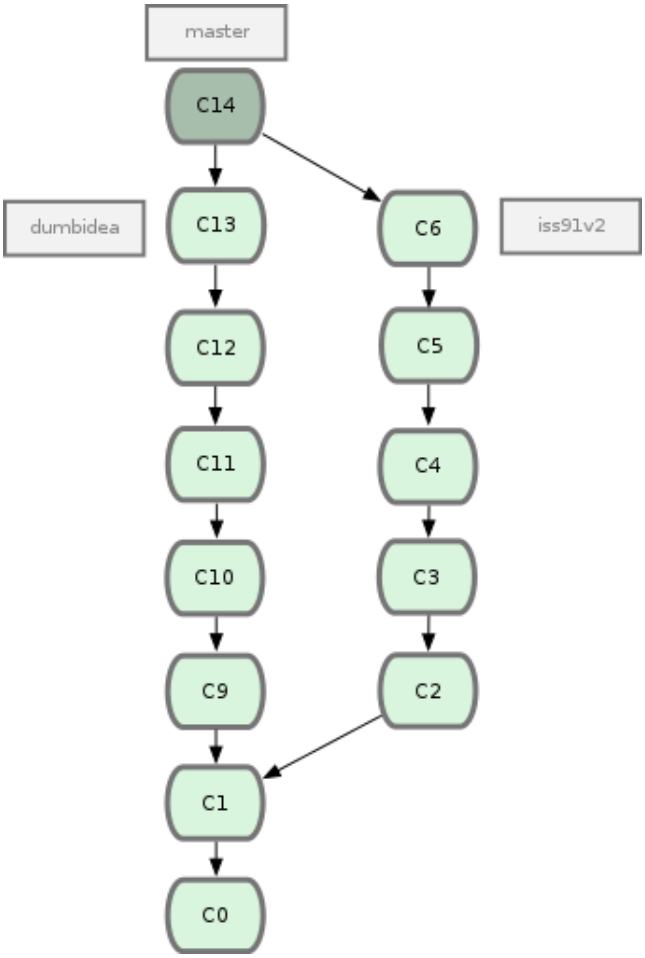

Figure 3.21: Your history after merging in dumbidea and iss91v2.

It's important to remember when you're doing all this that these branches are on your local computer. When you're branching and merging, everything is being done only in your Git repository — no communication with a remote server is happening.

3.5 Remote Branches

Remote branches are branches in remote repositories. They act like local branches that you yourself can't move but rather Git adjusts automatically whenever you communicate with a remote repository whose branches have changed since the last time you communicated with it. Remote branches act as bookmarks to remind you where the branches on remote repositories were the last time you communicated with them.

A remote branch name takes the form `(remote)/(branch)`. For instance, to refer to what the `master` branch on the `origin` remote looked like the last time you communicated with it, use the name `origin/master`. Let's say you're working on issue #53 with a partner and they created a branch named `iss53` that they pushed to the `origin` server. To refer to the branch on the server, use the name `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 named `git.ourcompany.com`. If you clone from this, Git automatically creates the name `origin` in your Git repository, copies all the data from `git.ourcompany.com` into your Git repository, and creates a pointer named `origin/master` in your repository pointing to where the `master` branch was in the remote repository when you did the clone. You can't move `origin/master` yourself. Git also creates your own `master` branch starting at the same place as origin's `master` branch, so you have something to work from (see Figure 3-22).

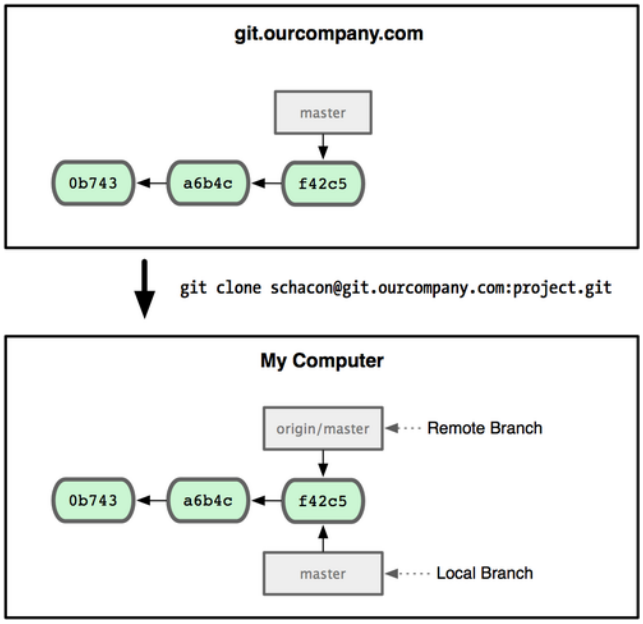

Figure 3.22: A Git clone gives you your own master branch and origin/master pointing to origin's master branch.

If you do some work on your local master branch, and, in the meantime, someone else changes the master branch on `git.ourcompany.com`, then the commit histories in the two Git repositories move forward differently. As long as you don't contact `git.ourcompany.com`, `origin/master` in your local Git repository doesn't move (see Figure 3-23).

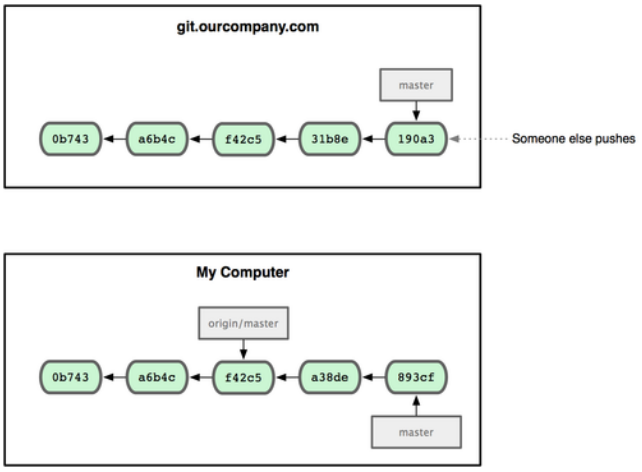

Figure 3.23: Working locally and having someone push to your remote server makes each history move forward differently.

To synchronize your work, run `git fetch origin`. This command fetches any data from the `origin` server (in this case `git.ourcompany.com`) that you don't yet have in your Git repository, and then updates your Git repository by moving `origin/master` to its new position (see Figure 3-24).

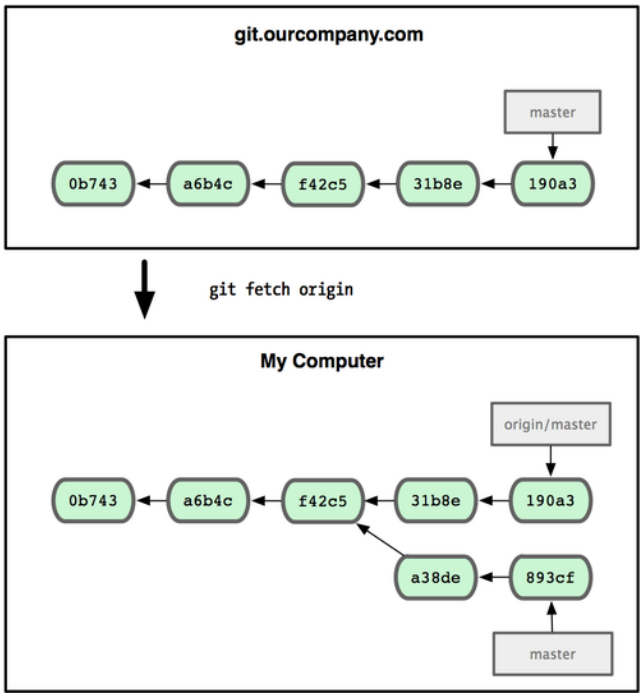

Figure 3.24: The git fetch command updates your remote references.

To demonstrate having multiple remote servers and what remote branches for those remote projects look like, let's assume `git.team1.ourcompany.com` is another Git server that's used only by one of your sprint teams. Add it as a new remote reference to the project you're currently working on by running `git remote add` as I described in Chapter 2. Name this remote `teamone`, which will be your shortname for that remote URL (see Figure 3-25).

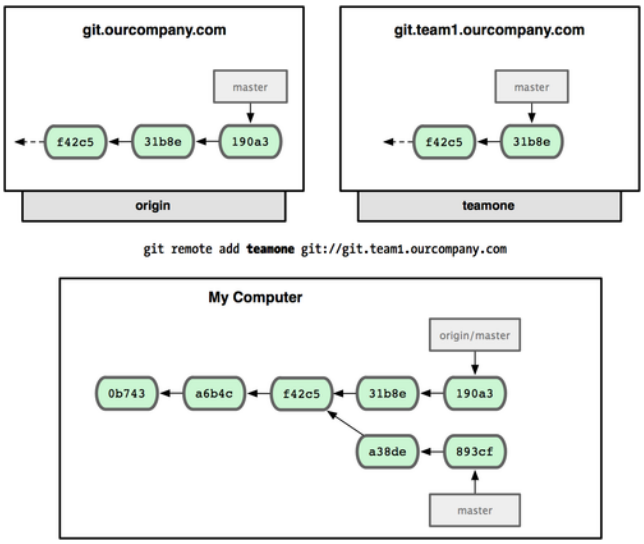

Figure 3.25: Adding another server as a remote.

Now, run `git fetch teamone` to fetch everything on the remote `teamone` server that you

don't already have. Because that server only has a subset of the data on the `origin` server right now, Git fetches no data but creates a remote branch pointer called `teamone/master` in your Git repository pointing to the commit that `teamone` has as its `master` branch (see Figure 3-26).

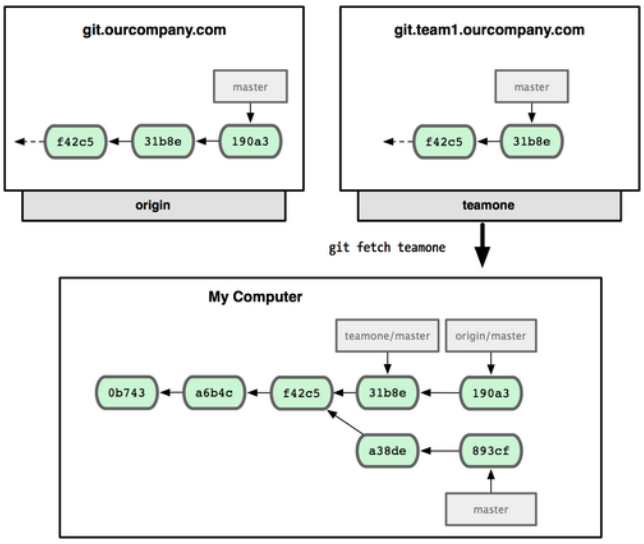

Figure 3.26: You get a reference to teamone's master branch position locally.

3.5.1 Pushing

To share a branch in your Git repository, push it to a remote Git repository that you have write access to. Keep in mind that your local branches aren't automatically synchronized with the remote branches they came from — you have to explicitly push the branches you want to share. That way, you can keep some branches private for work you don't want to share, and push only the branches you want to collaborate on.

If you have a branch named `serverfix` that you want to work on with others, push it the same way you pushed your first branch. Run `git push (remote) (branch)`.

```
$ git push origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

This takes your local `serverfix` branch and pushes it to update the remote `serverfix` branch. If you don't want the remote branch to be called `serverfix`, you could instead run `git push origin serverfix:awesomebranch` to push your local `serverfix` branch to the `awesomebranch` branch on the remote Git repository.

The next time one of your collaborators fetches from the remote server, they will get a reference called `origin/serverfix` to where the remote server's version of `serverfix` is.

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

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

To merge this work into your current working branch, run `git merge origin/serverfix`. If you want your own `serverfix` branch to work on, base it off your remote branch.

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

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

3.5.2 Tracking Branches

Checking out a local branch from a remote branch automatically creates what is called a *tracking branch*. Tracking branches are local branches that have a direct relationship to a remote branch. If you're on a tracking branch and type `git push`, Git automatically knows which server and branch to push to. Also, running `git pull` while on one of these branches fetches all the remote references and then automatically merges in the corresponding remote branch.

When you clone a repository, Git automatically creates a `master` branch that tracks `origin/master`. That's why `git push` and `git pull` work out of the box with no other arguments. However, you can set up other tracking branches — ones that don't track branches on `origin` and don't track the `master` branch. The simple case is the example you just saw, running `git checkout --track [branch] [remotename]/[branch]`.

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

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

```
$ git checkout --track -b sf origin/serverfix
Branch sf set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "sf"
```

Now, your `sf` local branch will automatically push to and pull from `origin/serverfix`.

3.5.3 Deleting Remote Branches

Suppose you're done with a remote branch — say, you and your collaborators are finished with a feature and have merged it into your remote's `master` branch. You delete a remote branch using the rather obtuse syntax `git push [remotename] :[branch]`. To delete your `serverfix` branch from your Git repository, run

```
$ git push origin :serverfix
To git@github.com:schacon/simplegit.git
 - [deleted]         serverfix
```

Boom. No more `serverfix` branch on the remote server. You may want to dog-ear this page, because you'll need that command, and you'll likely forget the syntax. A way to remember this command is by recalling the `git push [remotename] [localbranch]:[remotebranch]` syntax that I went over a bit earlier. If you leave off the `[localbranch]` portion, then you're basically saying, "Take nothing on my side and make it be `[remotebranch]`".

3.6 Rebasing

There are two main ways to integrate changes from one branch into another: `merging` and `rebasing`. You've already learned about merging. In this section you'll learn what rebasing is, how to do it, why it's a pretty amazing tool, and in what cases not to do it.

3.6.1 Basic Rebasing

If you go back to an earlier example from the Merge section (see Figure 3-27), you see that you diverged your work and made commits on two different branches.

The easiest way to integrate the branches, as I've already covered, is the `git 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 commit (C5), as shown in Figure 3-28.

However, there's another way: take the change that was introduced in C3 and reapply it on top of C4. In Git, this is called *rebasing*. The `git rebase` command takes all the changes that were committed on one branch and replays them on another.

In this example, run

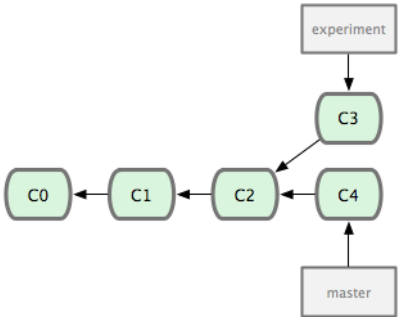

Figure 3.27: Your initial diverged commit history.

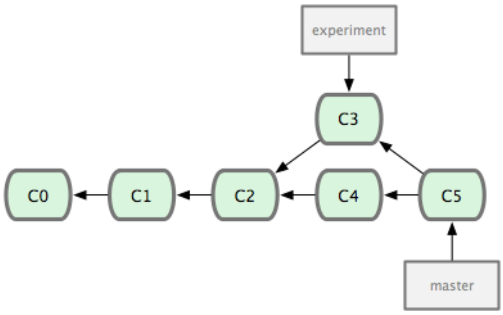

Figure 3.28: Merging a branch to integrate the diverged work history.

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

This works by going to the common ancestor (C2) of the branch you're on (`experiment`) and the branch you're rebasing onto (`master`), getting the diffs introduced by each commit (C3) of the branch you're on, saving those diffs to temporary files, resetting the current branch (`experiment`) to the same commit as the branch you're rebasing onto (C4), and finally applying each change in turn to the branch you're rebasing onto (`master`). Figure 3-29 illustrates this process.

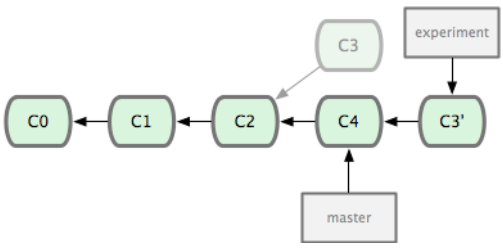

Figure 3.29: Rebasing the change introduced in C3 onto C4.

At this point, you can go back to the master branch and do a fast-forward merge (see Figure 3-30).

The snapshot pointed to by C3' is exactly the same as the one that was pointed to by C5 in the merge example. There's no difference in the end product of the integration, but

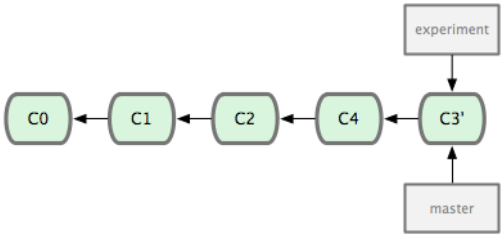

Figure 3.30: Fast-forwarding the master branch.

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.

3.6.2 More Interesting Rebases

Your rebase can also replay on something other than the rebase branch. Take a commit history like Figure 3-31, for example. You created a topic branch (`server`) to add some server-side functionality to your project, and made a commit (C3). Then, you branched off that to make the client-side changes (`client`) and committed a few times (C4 and C5). Finally, you went back to your server branch and did a few more commits (C6 and C7). Finally, you made a few commits on `master` (C8 and C9).

Suppose you decide to merge your client-side changes into your master branch for a release, but you want to hold off merging the server-side changes until they're tested further. You can take the changes on `client` that aren't on `server` (C4 and C5) and replay them onto `master` by running `git rebase --onto`.

```
$ git rebase --onto master server client
```

This basically says, "Check out the `client` branch, figure out the patches from the common ancestor of the `client` and `server` branches, and then replay the patches onto `master`." It's a bit complex but the result, shown in Figure 3-32, is pretty cool.

Now fast-forward your `master` branch (see Figure 3-33).

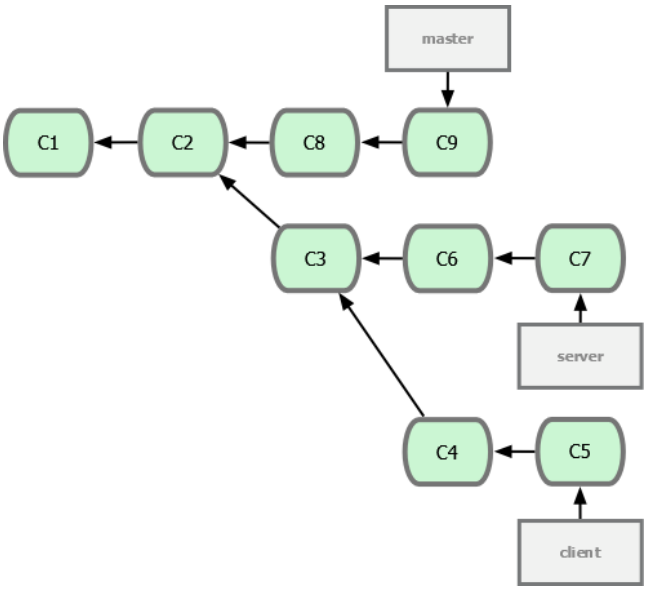

Figure 3.31: A history with a topic branch off another topic branch.

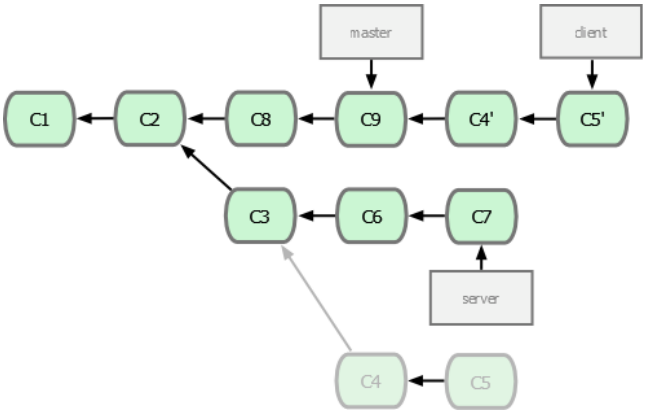

Figure 3.32: Rebasing a topic branch off another topic branch.

```
$ git checkout master
$ git merge client
```

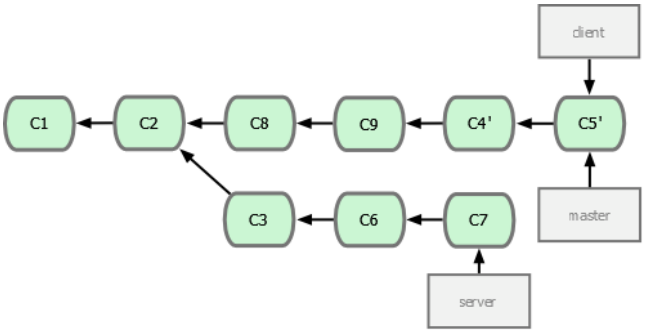

Figure 3.33: Fast-forwarding your master branch to include the client branch changes.

Let's say you decide to do the same thing with `server`. Rebase `server` onto `master` without having to check out `master` first by running `git rebase [basebranch] [topicbranch]` — which checks out the topic branch (in this case, `server`) 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 Figure 3-34.

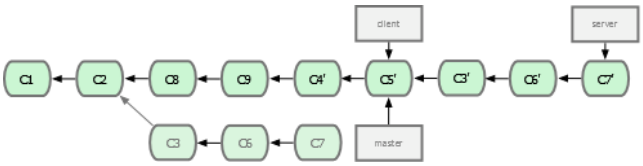

Figure 3.34: Rebasing your server branch on top of your master branch.

Then, fast-forward the base branch (`master`).

```
$ git checkout master
$ git merge server
```

At this point, remove the `client` and `server` branches because all the work is contained in `master` so you don't need them anymore. This leaves your history looking like Figure 3-35.

```
$ git branch -d client
$ git branch -d server
```

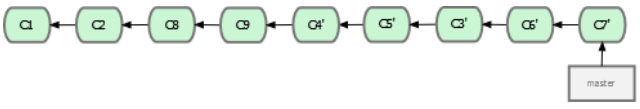

Figure 3.35: Final commit history.

3.6.3 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 you have pushed to a public repository.

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, you're abandoning existing commits and creating new ones that are similar but not exactly the same. If you push commits somewhere and other people pull

them and base work on them, and then you rewrite those commits with `git rebase` and push them again, your collaborators will have to re-merge their work. Things will get messy when you try to pull their work back into yours.

Let's look at an example of how making rebasing public can cause problems. Suppose you clone from a central server and then do some work. Your commit history looks like Figure 3-36.

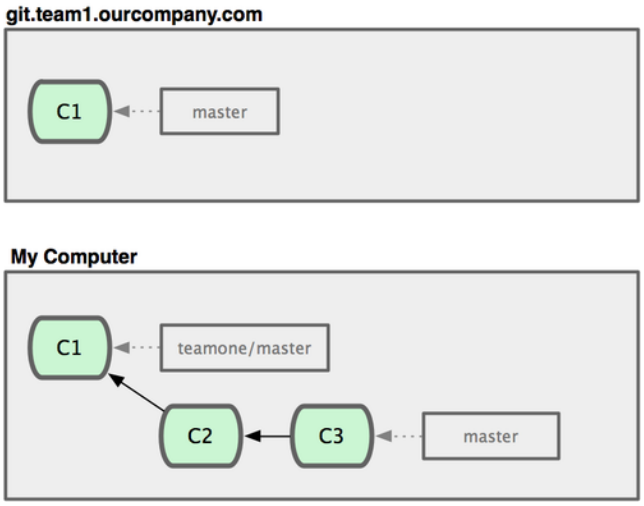

Figure 3.36: Clone a repository, and base some work on it.

Now, someone else does more work that includes a branch and merge, and pushes that work to the central server. Fetch and merge the new remote branch into your work, making your history look something like Figure 3-37.

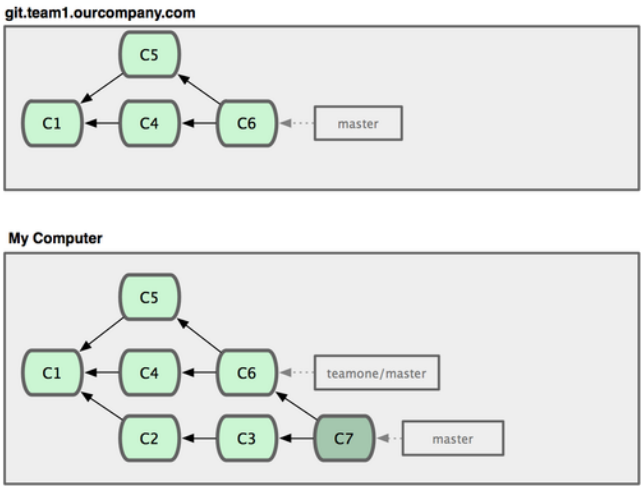

Figure 3.37: Fetch more commits, and merge them into your work.

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.

At this point, you have to merge this work in again, even though you've already done so. Rebasing changes the SHA-1 hashes of these commits so to Git they look like new commits, when in fact you already have the C4 work in your history (see Figure 3-39).

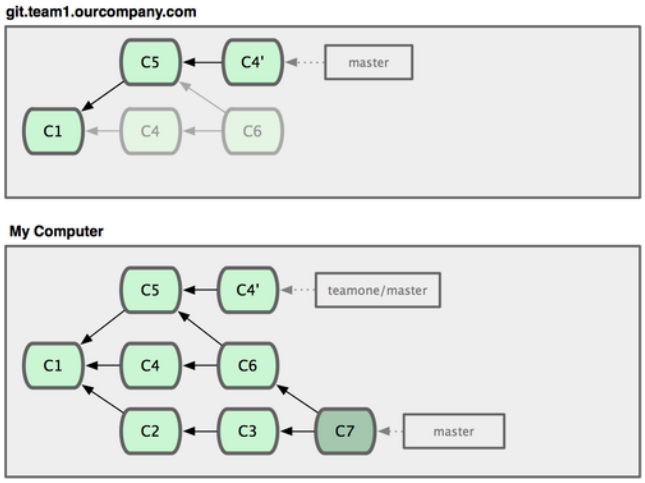

Figure 3.38: Someone pushes rebased commits, abandoning commits you've based your work on.

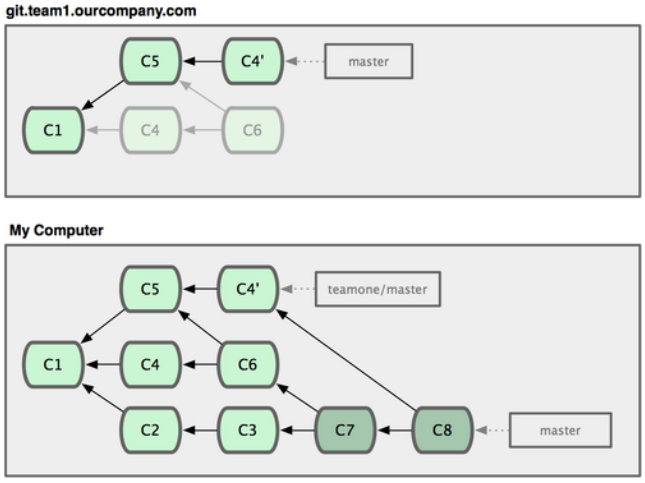

Figure 3.39: You merge in the same work again into a new merge commit.

You have to merge that work in at some point so you can keep up with the other developers in the future. After you do that, your commit history will contain both the C4 and C4' commits, which have different SHA-1 hashes but introduce the same work and have the same commit message. If you run `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 to the server, you'll reintroduce all those rebased commits to the central server, which can further confuse people.

If you treat rebasing as a way to work with commits before you push them, and if you only rebase commits that have never been available publicly, then you'll 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.

3.7 Summary

I've covered basic branching and merging in Git. You should feel comfortable creating and switching to new branches, switching between branches, and merging local branches.

You should also be able to share your branches by pushing them to a shared server, working with others on shared branches, and rebasing your branches before they're shared.

Chapter 4

Git on the Server

At this point, you should be able to run most of the day-to-day Git commands you'll need. However, in order to do any collaboration with Git, you'll need to access a remote Git repository. Although you can technically push changes to and pull changes from individuals' repositories, doing so is discouraged because that can fairly easily complicate what they're working on if you're not careful. Furthermore, collaborators want to be able to access each other's repository even if their computer is offline. So, having everyone share a repository is a better solution. Therefore, the preferred method for collaborating is to set up a repository that everyone has access to, and push to and pull from it. We'll refer to this repository as a "Git server" but since it generally takes a tiny amount of resources to host a Git repository, you'll rarely need to use a dedicated server as a Git server.

Running a Git server is simple. First, you choose which protocols you want your server to support. The first section of this chapter covers the available protocols and their pros and cons. The next sections explain some typical configurations using those protocols. Last, I'll go over a few hosted options, if you don't mind hosting your code on someone else's server and don't want to go through the hassle of setting up and maintaining your own server.

If you have no interest in running your own server, skip to the last section of this chapter to see some options for setting up a hosted account. Then, move on to the next chapter, where I discuss the ins and outs of working in a distributed source control environment.

A remote repository is generally a *bare repository* — a Git repository that has no working directory. Because the repository is only used as a collaboration point, there's no reason to have a working directory. The Git data is all that's necessary. In the simplest terms, a bare repository is a project's `.git` directory and nothing else.

4.1 The Protocols

Git can use four network protocols to transfer data: Local, Secure Shell (SSH), Git, and HTTP. Here I'll discuss what they are and when you'd want (or not want) to use them.

It's important to note that with the exception of HTTP, all of these protocols require Git to be installed on the server.

4.1.1 Local Protocol

The most basic protocol is the *local* protocol, in which the remote repository is in a directory on your local computer. This is often used if everyone on your team has access to a shared filesystem, such as an NFS mount, or in the less likely case that everyone logs in to the same computer. This isn't a good idea because placing everyone's code repository on the same disk makes a catastrophic loss much more likely.

If you have a shared filesystem mounted locally then you can clone, push to, and pull from a repository located on it. To clone from a local repository or to add one as a remote to an existing project, use the path to the repository in place of the URL. For example, to clone a local repository, run something like

```
$ git clone /opt/git/project.git
```

or

```
$ git clone file:///opt/git/project.git
```

Git operates slightly differently if you explicitly specify `file://` at the beginning of the URL than if you just give a local path. If you just specify the path, Git tries to use hardlinks or directly copy the files it needs. If you specify `file://`, Git uses the same technique that it normally uses to transfer data over a network, which is generally a lot less efficient. The main reason to specify the `file://` prefix is if you want a clean copy of the repository with no extraneous references or objects — like after an import from another version-control system or something similar (see Chapter 9). I'll use a normal path here because this is almost always faster.

To add a local repository to an existing Git project, run something like

```
$ git remote add local_proj /opt/git/project.git
```

Then, push to and pull from that remote as though you were transferring over a network.

The Pros

The pros of file-based repositories are that they're simple and use existing file permissions. If you already have a shared filesystem to which your whole team has access, setting up a repository is very easy. Stick the bare repository somewhere where everyone has access and set the read/write permissions as you would for any other shared directory. I'll discuss how to export a bare repository for this purpose in the next section, "Getting Git on a Server."

This is also a nice option for quickly pulling from someone else's working repository. If you and a co-worker are working on the same project and you want to pull from their repository, running a command like

```
$ git pull /home/john/project
```

is often easier than your co-worker first pushing to a remote server and then you pulling from there.

The Cons

The con of this method is that shared access is generally more difficult to set up and reach from multiple locations than basic network access. To push from your laptop when you're at home, you'd have to mount the remote disk containing the repository, which can be difficult and slow compared to the other network-based protocols.

It's also important to mention that this isn't necessarily the fastest option. A local repository is fast only if you have fast access to the repository. Accessing a repository via NFS is often slower than accessing the repository over SSH on the same remote server.

4.1.2 The SSH Protocol

Probably the most common transport protocol for Git is SSH. This is because SSH access to servers is already set up almost everywhere — and if it isn't, it's easy to do. SSH is also the only network-based protocol that you can use for both read and write access. The other two network protocols (HTTP and Git) are generally read-only, so even if they're available for the unwashed masses, you still need SSH for write access. 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, specify an ssh:// URL.

```
$ git clone ssh://user@server/project.git
```

Or, use the shorter SCP-like syntax for the SSH protocol.

```
$ git clone user@server:project.git
```

If you don't specify a user, Git substitutes your current username.

The Pros

The pros of using SSH are many. You have no choice if you want authenticated write access to a repository over the network. Second, SSH is relatively easy to set up — SSH daemons are commonplace, many network admins have experience with them, and many OS distributions include them and have tools to manage them. Also, access over SSH is secure — all data transfer is encrypted and authenticated. Last, like the Git and local protocols, SSH is efficient, compressing the data before transferring it.

The Cons

The negative aspect of SSH is that you can't provide anonymous read-only access to your repository using it. Users must have SSH access to your server to access your repository, even in read-only mode, which makes SSH access inappropriate for open source projects. If you're using it only within your corporate network, SSH may be the only protocol you need to deal with. To allow anonymous read-only access to your projects, you'll have to set up SSH for you to push over but something else for others to pull over.

4.1.3 The Git Protocol

Next is the Git protocol. This requires a special daemon that comes packaged with Git. It listens on a dedicated port (9418) that provides a service similar to SSH, but with absolutely no authentication. In order for a repository to be served using the Git protocol, you must create the `git-export-daemon-ok` file — the daemon won't serve a repository that doesn't contain that file. Other than that there's no security. Either the Git repository is available for everyone to clone or it isn't available to anyone. This means that you generally don't allow push access using this protocol. You can enable push access but, given the lack of authentication, if you do so, anyone on the internet who knows your project's URL could push to your project. Suffice it to say that this is rarely a good thing.

The Pros

The Git protocol is the fastest Git repository 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 way to access your project. Generally, you'll pair it with SSH access for the few developers who have push (write) access, leaving everyone else to use the Git protocol for read-only access. It's also probably the most difficult protocol to set up. It requires running a daemon — I'll describe setting one up in the "Gitosis" section of this chapter — and requires `xinetd` configuration or the like, which isn't always a walk in the park. It also requires your firewall to allow access to port 9418, which isn't a standard port that corporate firewalls allow.

4.1.4 The HTTP Protocol

Last, there's the HTTP protocol. (Since HTTPS works the same as HTTP as far as Git's concerned, I won't include it here). The beauty of the HTTP protocol is the simplicity of setting it up. Simply put the bare Git repository under your HTTP document root and set up a specific `post-update` hook, and you're done (See Chapter 7 for details on Git hooks). At

that point, anyone who can access the web server can also clone your repository. To allow read-only access to your repository over HTTP, do something like

```
$ 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. Then, other people can clone via something like

```
$ git clone http://example.com/gitproject.git
```

In this particular case, I'm using the `/var/www/htdocs` path that's common for Apache setups, but you can use any path — just put the bare repository in the path. The Git data is served as static files (see Chapter 9 for details about exactly how).

It's possible to make Git push over HTTP as well, although that technique isn't as widely used and requires setting up a complex WebDAV environment. Because of these issues, I won't cover it in this book. If you're interested in using the HTTP-push protocols, read about preparing a repository using these protocols at http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt. One nice thing about Git pushing over HTTP is that you can use any WebDAV server, without any specific Git requirements. This means you can use this approach if your web-hosting provider supports WebDAV for updating your web site.

The Pros

The upside of using the HTTP protocol is that it's so easy to set up. Running the handful of required commands gives you a simple way to give the world read-only access to your Git repository. The HTTP protocol also isn't very resource intensive on your server. Because Git repositories are static files, a normal Apache server can serve thousands of files per second on average. It's difficult to overload even a small server.

Another nice thing is that HTTP is such a commonly used protocol that corporate firewalls are often set up to pass HTTP traffic.

The Cons

The downside of serving your repository over HTTP is that it's relatively inefficient in terms of network resources. It generally takes a lot longer to clone or fetch from the repository, and there's a lot more network overhead and data transfer volume over HTTP than with

any of the other network protocols. For more information about the differences in efficiency between the HTTP protocol and the other protocols, see Chapter 9.

4.2 Getting Git on a Server

In order to initially set up a Git server, you have to start with a new bare repository — a repository that doesn't contain a working directory. This is generally straightforward to do. By convention, bare repository directories end in `.git`. To clone a repository to create a new bare repository, run `git clone --bare`.

```
$ git clone --bare my_project my_project.git
Initialized empty Git repository in /opt/projects/my_project.git/
```

The output for this command is a little confusing. Since `git clone` is basically a `git init` then a `git fetch`, you see some output from the `git init` part, which creates an empty directory. The actual object transfer generates no output. You should now have a copy of the Git repository in your `my_project.git` directory.

This is roughly equivalent to

```
$ cp -Rf my_project/.git my_project.git
```

There are a couple of minor differences but, as far as you're concerned, this is close to the same thing. It copies just the Git repository, without a working directory, into a new directory.

4.2.1 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 access protocols. Let's say you've set up a server called `git.example.com` that you have SSH access to, and you want to store all your Git repositories under the `/opt/git` directory. Set up your new repository by copying your bare repository over.

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

At this point, other users who have SSH and read access to the `/opt/git` directory on the server can clone your repository by running

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

If a user SSHs into a server and has write access to the `/opt/git/my_project.git` directory, they will also be able to push to the repository. As an aside, Git automatically adds group write permissions to a repository if you run `git init --shared`.

```
$ ssh user@git.example.com
$ cd /opt/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.

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 accounts on a server that users can SSH to, 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, using the Gitosis tool, and more. However, keep in mind that to collaborate with a couple of people on a private project, all you *need* is Git, an SSH server, and a bare repository.

4.2.2 Small Setups

If you're a small outfit or are just trying out Git in your organization and have only a few developers, setting up a Git server is easy. The most complicated task is user management. If you want some repositories to be read-only to certain users and read/write to others, access and permissions can be a bit difficult to configure.

SSH Access

If you already have a server to which all your developers have SSH access, it's generally easiest to set up your first repository there, because you have to do almost no work (as I covered in the last section). If you want more complex access controls on your repositories, use normal filesystem permissions.

To place your repositories on a server that doesn't already have accounts for everyone on your team, you must set up SSH access for them. I assume that you can already access the server using SSH yourself.

There are a few ways to 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` and set temporary passwords for every user.

A second method is to create a single 'git' user, ask every user who needs write access to send you their SSH public key, and add that key to the new 'git' user's `~/.ssh/authorized_keys` file. At that point, everyone will be able to access that machine as the 'git' user. This doesn't affect the commit data in any way — the SSH user you connect as doesn't affect user that Git sees as doing the commits.

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 Git server, any SSH authentication mechanism you can think of should work.

4.3 Generating Your SSH Key Pair

Many Git servers authenticate using SSH key pairs. Each user on your system must generate a key pair. The process for doing so is similar across systems using SSH. First, make sure the user doesn't already have a key pair. By default, a user's SSH keys are stored in their `~/.ssh` directory. Check for SSH keys by listing the contents of that directory.

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa       known_hosts
config            id_dsa.pub
```

Look for a pair of files usually named `something` and `something.pub`, where `something` is usually `id_dsa` or `id_rsa`. The `.pub` file contains the public key, and the other file contains the private key. If you don't see these files (or if you don't even see a `.ssh` directory), the user can create them by running `ssh-keygen`, which is provided with the OpenSSH package on Linux/Mac systems and comes with the MSysGit package on Windows.

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local
```

First, `ssh-keygen` confirms where to save the key (`.ssh/id_rsa`), and then asks twice for a passphrase, which can be left empty if the user doesn't want to type a passphrase when using the key pair.

Now, each user that does this has to send their public key (the `.pub` file) to whoever is administering the Git server (assuming you're using an SSH server setup that requires public keys). Public keys look something like

```
$ 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@agadorlaptop.local
```

As the name implies, public keys are indeed public, so there's no danger including them in email messages. You could even print them on the front page of your local newspaper and nothing bad would happen. For a more in-depth tutorial on creating SSH keys on multiple operating systems, see the GitHub guide on SSH keys at http://github.com/guides/providing-your-ssh-key.

4.4 Setting Up the Server

Let's walk through setting up SSH access on the server. In this example, you'll use the `authorized_keys` method for authenticating your users. I also assume you're running a standard Linux distribution, like Ubuntu. First, create a 'git' user and an `.ssh` directory for that user.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```

Next, add the SSH public keys for everyone who'll be connecting using the 'git' user account to the `authorized_keys` file for the 'git' user. Let's assume you've received a few public keys by e-mail and saved them in temporary files. Again, the public keys look something like

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Just append them to the 'git' user's `authorized_keys` file.

```
$ 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, create an empty repository by running `git init --bare`, which initializes a repository without a working directory.

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git --bare init
```

Then, John, Josie, or Jessica can push the first version of their project into the new repository by adding it as a remote and pushing to it.

Note that someone must login to the server and create a bare repository every time you want to add a project. Let's use `gitserver` as the hostname of the Git server. If you created a DNS entry for `gitserver`, then use the following commands:

```
# on Johns computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

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

```
# on others' computers
$ git clone git@gitserver:/opt/git/project.git
$ cd project
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

This method quickly provisions a read/write Git server for a handful of developers.

As an extra precaution, you can easily restrict the 'git' user to only running Git commands by using a limited shell called `git-shell` that comes with Git. If you set this as the 'git' user's login shell, then the 'git' user doesn't get normal shell access on the server. To

do this, specify `git-shell` instead of bash or csh as your user's login shell. To do so, you'll likely have to edit your `/etc/passwd` file.

```
$ sudo vim /etc/passwd
```

You should find a line that looks something like

```
git:x:1000:1000::/home/git:/bin/sh
```

Change `/bin/sh` to `/usr/bin/git-shell` (or run `which git-shell` to see where `git-shell` is installed and use that path). The edited line should look something like

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

Now, the 'git' user can only use the SSH connection to push and pull Git repositories and can't login to the machine. If they try, they'll see a login rejection like

```
$ ssh git@gitserver
fatal: What do you think I am? A shell?
Connection to gitserver closed.
```

4.5 Public Access

What if you want to provide anonymous read access to your project? Perhaps instead of hosting an internal private project, you want to host an open source project. Or maybe you have a bunch of automated build servers or continuous integration servers that change a lot, and you don't want to have to generate SSH keys all the time — you just want to allow simple anonymous read access.

Probably the simplest way for smaller setups to accomplish this is to run a static web server with its document root where your Git repository is, with the `post-update` hook that I mentioned in the first section of this chapter enabled. Let's work from the previous example. Say you have your repository in the `/opt/git` directory, and an Apache server is running on your server. Again, you can use any web server for this but, as an example, I'll show a basic Apache configuration that should give you an idea of what you might need.

First, you need to enable the hook.

```
$ cd project.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

What does this `post-update` hook do? It looks basically like

```
$ cat .git/hooks/post-update
#!/bin/sh
exec git-update-server-info
```

This means that when you push to the server via SSH, Git runs `git-update-server-info` to update the repository so that others can pull over HTTP.

Next, add a VirtualHost entry to your Apache configuration with the root directory of your Git projects as the DocumentRoot. Here, I're assuming that you have DNS set up to direct `*.gitserver` to the server you're using to run all this.

```
<VirtualHost *:80>
    ServerName git.gitserver
    DocumentRoot /opt/git
    <Directory /opt/git/>
        Order allow, deny
        allow from all
    </Directory>
</VirtualHost>
```

You'll also need to set the Unix group of the `/opt/git` directory to `www-data` so your web server has read-access to the repository. That's the group the Apache instance running the CGI script will (by default) be running as.

```
$ chgrp -R www-data /opt/git
```

After restarting Apache, you should be able to clone your repository by specifying the URL for your project.

```
$ git clone http://git.gitserver/project.git
```

Using this method you can set up HTTP-based read access to any of your Git projects in a few minutes. Another simple option for public unauthenticated access is to start a Git daemon. I'll cover this option in the next section, if you prefer that route.

4.6 GitWeb

Now that you've configured read/write and read-only access to your project, you can set up a simple web-based repository visualizer. Git comes with a CGI script called GitWeb that's commonly used for this. See GitWeb in use at sites like `http://git.kernel.org` (see Figure 4-1).

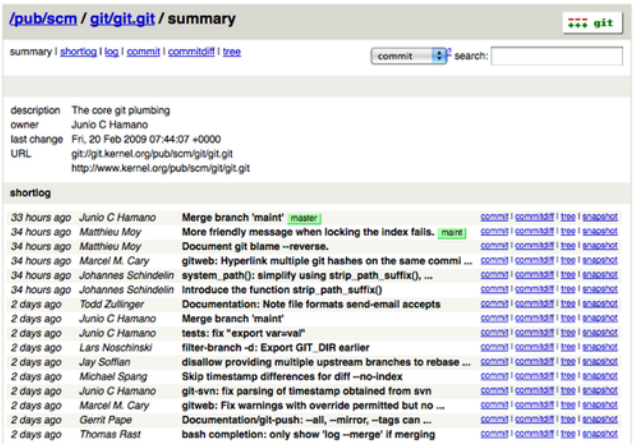

Figure 4.1: The GitWeb web-based user interface.

To check out what GitWeb would look like for your project, Git comes with a command to start a temporary web server if you have a lightweight web server already installed on your system, such as `lighttpd` or `webrick`. `lighttpd` is often installed on Linux machines, so you may be able to get it to run by running `git instaweb` in your project directory. If you're running a Mac, OS X comes preinstalled with Ruby, so `webrick` may be your best bet. To tell `instaweb` to use something other than `lighttpd`, run it with the `--httpd` option.

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

This starts a web server on port 1234 and then automatically starts a web browser that opens on the GitWeb page. This is all you have to do. When you're done and want to shut down the server, run the same command with the `--stop` option.

```
$ git instaweb --httpd=webrick --stop
```

To run the web interface all the time, set up the CGI script that creates the output so that it's run by your normal web server. Some Linux distributions have a `gitweb` package that you may be able to install via `apt` or `yum`, so you may want to try that first. I'll walk though installing GitWeb manually very quickly. First, get the Git source code, which includes GitWeb, and generate the custom CGI script.

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" \
        prefix=/usr gitweb
$ sudo cp -Rf gitweb /var/www/
```

Notice that you have to tell the `make` command where to find your Git repositories by assigning a value to the `GITWEB_PROJECTROOT` environment variable. Now, make Apache use CGI to run that script. To do this, add a VirtualHost section.

```
<VirtualHost *:80>
    ServerName gitserver
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
        Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

Again, GitWeb can be run by any CGI capable web server. If you prefer something else, it shouldn't be difficult to set up. At this point, you should be able to visit `http://gitserver/` to view your repositories online, and use `http://git.gitserver` to clone and fetch your repositories over HTTP.

4.7 Gitosis

Keeping all users' public keys in the `authorized_keys` file for limiting access works well only up to a point. When you have hundreds of users, this technique becomes painful. Making a change requires logging in to the Git server. Plus, there's no access control — everyone whose public key is in `authorized_keys` has read and write access to every project.

At this point, you may consider turning to a widely used software project called Gitosis. This is basically a set of scripts that helps manage the `authorized_keys` file and implements simple access controls. The really interesting part is that the UI for this tool isn't a web interface, but rather a special Git repository. You set up user access and control in that repository so when you push it, Gitosis reconfigures the Git server based on what's in the repository. This is very clever.

Installing Gitosis isn't the simplest task ever, but it's not too difficult. It's easiest to run Gitosis on a Linux server — these examples use a stock Ubuntu 8.10 server.

Gitosis requires the Python setuptools package, which Ubuntu calls python-setuptools.

```
$ apt-get install python-setuptools
```

Next, clone and install Gitosis from the Gitosis project's main site.

```
$ git clone git://eagain.net/gitosis.git
$ cd gitosis
$ sudo python setup.py install
```

That installs several executables for Gitosis to use. Gitosis expects its repositories to be under `/home/git`. But you've already set up your repositories in `/opt/git`. Instead of reconfiguring everything, create a symlink.

```
$ ln -s /opt/git /home/git/repositories
```

Gitosis is going to manage your SSH keys, so remove the current `authorized_keys` file, and let Gitosis control it automatically. For now, move it out of the way.

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

Next, restore the normal shell for the 'git' user if you've changed the shell to `git-shell`. Users still won't be able to log in, but Gitosis will take care of that. So, change the line in your `/etc/passwd` file

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

back to

```
git:x:1000:1000::/home/git:/bin/sh
```

Now, it's time to initialize Gitosis. Run the `gitosis-init` command with the public key of the user you want to manage Gitosis as input.

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /opt/git/gitosis-admin.git/
Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

This lets the user whose public key you gave modify the main Git repository that controls the Gitosis setup. Next, set the execute bit on the `post-update` script in your new control repository.

```
$ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

You're ready to roll. If you're set up correctly, try to SSH into your Git server as the user whose public key you added when initializing Gitosis. You should see something like

```
$ ssh git@gitserver
PTY allocation request failed on channel 0
fatal: unrecognized command 'gitosis-serve schacon@quaternion'
  Connection to gitserver closed.
```

That means Gitosis recognized you but blocked you because you're not trying to run any Git commands. So, let's run an actual Git command to clone the Gitosis repository.

```
# on your local computer
$ git clone git@gitserver:gitosis-admin.git
```

Now you have a directory named `gitosis-admin`, which has two parts.

```
$ cd gitosis-admin
$ find .
./gitosis.conf
./keydir
./keydir/scott.pub
```

`gitosis.conf` is the control file that specifies users, repositories, and permissions. The `keydir` directory is where you store the public keys of all the users who need access to your repositories — one file per user. The name of the file in `keydir` (`scott.pub` above) will be different for you — Gitosis uses the name from the description at the end of the public key that was imported with the `gitosis-init` script.

If you look at `gitosis.conf`, it should only specify information about the `gitosis-admin` project that you just cloned.

```
$ cat gitosis.conf
[gitosis]
```

```
[group gitosis-admin]
writable = gitosis-admin
members = scott
```

It shows that 'scott' — the user whose public key you used to initialize Gitosis — is the only user with access to the `gitosis-admin` project.

Now, let's add a new project. Add a new section called `mobile` which lists the developers on your mobile team and projects that they need access to. Because 'scott' is the only user in the system right now, add him as the only member, and create a new project called `iphone_project` to start on.

```
[group mobile]
writable = iphone_project
members = scott
```

Whenever you make changes to the `gitosis-admin` project, commit the changes and push to the Git server in order for the changes to take effect.

```
$ git commit -am 'add iphone_project and mobile group'
[master]: created 8962da8: "changed name"
 1 files changed, 4 insertions(+), 0 deletions(-)
$ git push
Counting objects: 5, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 272 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@gitserver:/opt/git/gitosis-admin.git
   fb27aec..8962da8  master -> master
```

Make your first push to the new `iphone_project` project by adding your Git server as a remote to your local version of the project and then pushing to the remote. You no longer have to manually create a bare repository for new projects on the server — Gitosis creates them automatically when it sees the first push.

```
$ git remote add origin git@gitserver:iphone_project.git
$ git push origin master
Initialized empty Git repository in /opt/git/iphone_project.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 230 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
```

```
To git@gitserver:iphone_project.git
 * [new branch]      master -> master
```

Notice that you don't need to specify the path (in fact, doing so won't work), just a colon and then the name of the project — Gitosis finds the path for you.

You want to work on this project with your friends, so you'll have to re-add their public keys. But instead of appending them manually to the `~/.ssh/authorized_keys` file on your server, add them, one key per file, into the `keydir` directory. How you name the keys determines how you refer to the users in `gitosis.conf`. Let's re-add the public keys for John, Josie, and Jessica.

```
$ cp /tmp/id_rsa.john.pub keydir/john.pub
$ cp /tmp/id_rsa.josie.pub keydir/josie.pub
$ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub
```

Now, add them all to your 'mobile' team so they have read and write access to `iphone_project`.

```
[group mobile]
writable = iphone_project
members = scott john josie jessica
```

After you commit and push that change, all four users will have read and write access to that project.

Gitosis has simple access controls as well. For John to have only read access to `iphone_project`, do this instead.

```
[group mobile]
writable = iphone_project
members = scott josie jessica

[group mobile_ro]
readonly = iphone_project
members = john
```

Now John can clone the project and get updates, but Gitosis won't allow him to push to the project. Create as many of these groups as you want, each containing different users and projects. You can also specify another group as a member (using `@` as prefix), to include all of the members automatically.

```
[group mobile_committers]
members = scott josie jessica

[group mobile]
writable  = iphone_project
members   = @mobile_committers

[group mobile_2]
writable  = another_iphone_project
members   = @mobile_committers john
```

If you have any problems, add `loglevel=DEBUG` in the `[gitosis]` section. If you've lost push access by pushing a messed-up configuration, manually fix the file on the server under `/home/git/.gitosis.conf` — the file from which Gitosis reads its configuration. A push to the project takes the `gitosis.conf` file you just pushed and sticks it there. If you edit that file manually, it retains your modifications until the next successful push to the `gitosis-admin` project.

4.8 Gitolite

This section serves as a quick introduction to Gitolite, and provides basic installation and setup instructions. It can't, however, replace the enormous amount of documentation that comes with Gitolite . There may also be occasional changes to this section itself, so you may also want to look at the latest version of the documentation here.

Gitolite is an authorization layer on top of Git, relying on `sshd` or `httpd` for authentication. (Recap: authentication is identifying who the user is, authorization is deciding what he is allowed to do).

Gitolite allows specifying permissions not just by repository, but also by branch or tag names within each repository. That is, you can specify that certain people (or groups of people) can only push certain "refs" (branches or tags) but not others.

4.8.1 Installing

Installing Gitolite is very easy, even if you don't read the extensive documentation that comes with it. You need an account on a Unix server of some kind. You do not need root access, assuming Git, Perl, and an OpenSSH compatible SSH server are already installed. In the examples below, I'll use the `git` account on a host called `gitserver`.

Gitolite is somewhat unusual as far as "server" software goes — access is via SSH, and so every user on the Git server is a potential "gitolite host". I'll describe the simplest installation method in this article. For the other methods please see the documentation.

To begin, create a user called `git` on your Git server and login as this user. Copy your SSH public key from your workstation, naming it `<yourname>.pub` (I'll use `scott.pub` in the examples). Then, run these commands:

```
$ git clone git://github.com/sitaramc/gitolite
$ gitolite/install -ln
# assumes $HOME/bin exists and is in your $PATH
$ gitolite setup -pk $HOME/scott.pub
```

That last command creates a new Git repository called `gitolite-admin` on the Git server.

Finally, back on your workstation, run `git clone git@gitserver:gitolite-admin`. You're done! Gitolite has now been installed on the server, and you now have a brand new repository called `gitolite-admin` in your workstation. Administer your Gitolite setup by making changes to this repository and pushing.

4.8.2 Customizing the Install

While the default quick installation works for most people, there are some ways to customize it. You can make some changes by editing `gitolite.conf`, but if that isn't sufficient, there's documentation on customizing Gitolite.

4.8.3 Config File and Access Control Rules

Once the install is done, switch to the `gitolite-admin` clone you just made, and poke around to see what you have.

```
$ cd ~/gitolite-admin/
$ ls
conf/  keydir/
$ find conf keydir -type f
conf/gitolite.conf
keydir/scott.pub
$ cat conf/gitolite.conf

repo gitolite-admin
    RW+                 = scott

repo testing
    RW+                 = @all
```

Notice that "scott" (the name of the public key in the `gitolite setup` command you used earlier) has read-write permissions on the `gitolite-admin` repository as well as a public key file of the same name.

Adding users is easy. To add a user called "alice", obtain her public key, name it `alice.pub`, and put it in your `keydir` directory in the `gitolite-admin` repository you just made. Add, commit, and push the change, and "alice" has been added.

The config file syntax for Gitolite is well documented, so I'll only mention some highlights here.

You can group users or repositories for convenience. The group names are just like macros. When defining them, it doesn't even matter whether they're projects or users. That distinction is only made when you *use* the "macro".

```
@oss_repos      = linux perl rakudo git gitolite
@secret_repos   = fenestra pear

@admins         = scott
@interns        = ashok
@engineers      = sitaram dilbert wally alice
@staff          = @admins @engineers @interns
```

You can control permissions at the "ref" level. In the following example, interns can only push the "int" branch. Engineers can push any branch whose name starts with "eng-", and tags that start with "rc" followed by a digit. Admins can do anything (including rewind) to any ref.

```
repo @oss_repos
    RW  int$                = @interns
    RW  eng-                = @engineers
    RW  refs/tags/rc[0-9]   = @engineers
    RW+                     = @admins
```

The expression after the `RW` or `RW+` is a regular expression (regex) that the refname (ref) being pushed is matched against. So I call it a "refex"! Of course, a refex can be far more powerful than shown here, so don't overdo it if you're not comfortable with Perl regexes.

Also, as you probably guessed, Gitolite prefixes the refex with `refs/heads/` as a syntactic convenience if the refex doesn't begin with `refs/`.

An important feature of the config file's syntax is that all the rules for a repository need not be in one place. Keep all the common stuff together, like the rules for all `oss_repos` shown above, then add specific rules for specific cases later on, like

```
repo gitolite
    RW+                     = sitaram
```

That rule will just get added to the ruleset for the `gitolite` repository.

At this point you might be wondering how the access control rules are actually applied, so I'll go over that briefly.

There are two levels of access control in Gitolite. The first is at the repository level. If you have read (or write) access to *any* ref in the repository, then you have read (or write) access to the repository. This is the only access control that Gitosis had.

The second level, applicable only to "write" access, is by branch or tag within a repository. The username, the access being attempted (`W` or `+`), and the refname being updated are known. The access rules are checked in order of appearance in the config file, looking for a match for this combination (but remember that the refname is regex-matched, not merely string-matched). If a match is found, the push succeeds. A fallthrough results in access being denied.

4.8.4 Advanced Access Control with "deny" rules

So far, you've only seen `R`, `RW`, or `RW+` permissions. However, Gitolite allows another permission: `-`, standing for "deny". This gives a lot more power, at the expense of some complexity, because now fallthrough is not the *only* way for access to be denied, so the *order of the rules now matters*!

Let's say, in the situation above, I want engineers to be able to rewind any branch *except* `master` and `integ`. Here's how to do that:

```
    RW  master integ    = @engineers
    -   master integ    = @engineers
    RW+                 = @engineers
```

Again, simply follow the rules top down until you hit a match for your access mode, or a deny. Non-rewind push to `master` or `integ` is allowed by the first rule. A rewind push to those refs doesn't match the first rule, drops down to the second, and is therefore denied. Any push (rewind or non-rewind) to refs other than `master` or `integ` won't match the first two rules anyway, and the third rule allows it.

4.8.5 Restricting pushes by files changed

In addition to restricting the branches a user can push changes to, you can also restrict what files they're allowed to touch. For example, perhaps `Makefile` shouldn't be changed by just anyone, because a lot of things depend on it or would break if the changes are not done *just right*. Tell Gitolite.

```
repo foo
    RW                      =   @junior_devs @senior_devs

    -   VREF/NAME/Makefile  =   @junior_devs
```

Users migrating from older Gitolite versions should note that this feature's behavior has changed significantly. Please see the migration guide for details.

4.8.6 Personal Branches

Gitolite also has a feature called “personal branches” (or rather, “personal branch namespaces”) that can be very useful in a corporate environment.

A lot of code exchange in the Git world happens by “please pull” requests. In a corporate environment, however, unauthenticated access to a server isn't allowed, and developer workstations can't do authentication, so you have to push to a central server and then ask someone to pull from there.

This would normally cause the same branch name clutter as in a CVCS. Plus setting up permissions on the central server becomes a chore for the admin.

Gitolite lets you define a “personal” or “scratch” namespace prefix for each developer (for example, `refs/personal/<devname>/*`). See the documentation for details.

4.8.7 “Wildcard” repositories

Gitolite allows specifying repositories using wildcards (actually Perl regexes), like, for example `assignments/s[0-9][0-9]/a[0-9][0-9]`. It also allows assigning the permission mode (`C`) which permits creating repositories based on such wild cards, automatically assigning ownership to the specific user who created the repository, and allowing them to hand out `R` and `RW` permissions to other users to collaborate. Again, please see the documentation for details.

4.8.8 Other Features

I'll round off this discussion with a sample of other features, all of which, and many others, are described in great detail in the documentation.

Logging: Gitolite logs all successful accesses. If you're somewhat relaxed about giving people rewind permissions (`RW+`) and some kid blows away `master`, the log file is a life saver by easily and quickly showing the SHA-1 hash that got hosed.

Access rights reporting: Another convenient feature is what happens when you try to just ssh to the server. Gitolite shows what repositories you have access to, and what access you have. Here's an example.

```
hello scott, this is git@git running gitolite3 v3.01-18-g9609868 on git 1.7.4.4

     R     anu-wsd
     R     entrans
     R  W  git-notes
     R  W  gitolite
     R  W  gitolite-admin
     R     indic_web_input
     R     shreelipi_converter
```

Delegation: In really large installations, Gitolite can delegate responsibility for groups of repositories to various administrators to manage independently. This reduces the load on

the main administrator, and makes him less of a bottleneck.

Mirroring: Gitolite can help maintain multiple mirrors, and switch between them easily if the primary server goes down.

4.9 Git Daemon

For public unauthenticated read access to your projects, you'll want to move past the HTTP protocol and start using the Git protocol. The main reason is speed. The Git protocol is far more efficient and faster than the HTTP protocol.

Again, this is for unauthenticated read-only access. Running this on a server outside your firewall should only be done for projects that are publicly visible to the world. If the server you're running it on is inside your firewall, you might use it for projects that a large number of people or computers (continuous integration or build servers) have read-only access to, when you don't want to have to add an SSH key for each user.

In any case, the Git protocol is relatively easy to set up. Basically, run this command to start a Git daemon.

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

`--reuseaddr` allows the server to restart without waiting for old connections to time out, `--base-path` allows cloning without specifying a project's entire path, and the path at the end tells the Git daemon where to look for repositories to export. If you're behind a firewall, you must also open access to port 9418 on the box you're setting this up on.

You can daemonize this process a number of ways, depending on your operating system. On an Ubuntu machine, use an Upstart script. So, put this script in '/etc/event.d/local-git-daemon'.

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
    --user=git --group=git \
    --reuseaddr \
    --base-path=/opt/git/ \
    /opt/git/
respawn
```

For security reasons, you're strongly encouraged to run this daemon as a user with read-only permissions to the repositories. Do this by creating a new user 'git-ro' and running the daemon as this user. For the sake of simplicity I'll simply run it as the same 'git' user that Gitosis is running as.

When you restart your server, your Git daemon will start automatically and respawn if it goes down. To start the daemon without having to reboot, run

```
initctl start local-git-daemon
```

On other systems, you may want to use `xinetd`, a script in your `sysvinit` system, or something else — as long as you get the `git` command daemonized and set up to restart automatically.

Next, tell your Gitosis server which repositories to allow unauthenticated Git server-based access to. If you add a section for each repository, specify the repository your Git daemon should allow reading from. To allow Git protocol access for the `iphone_project`, add this to the end of the `gitosis.conf` file:

```
[repo iphone_project]
daemon = yes
```

When that's committed and pushed, your running daemon should start serving requests for the project to anyone who has access to port 9418 on your Git server.

If you decide not to use Gitosis, but you want to set up a Git daemon, run the following commands on each project the Git daemon should serve:

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

The presence of the `git-daemon-export-ok` file tells Git that it's OK to serve this project without authentication.

Gitosis can also control which projects GitWeb shows. First, add something like the following to `/etc/gitweb.conf`:

```
$projects_list = "/home/git/gitosis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

Control which projects GitWeb lets users browse by adding or removing a `gitweb` setting in the Gitosis configuration file. For instance, for `iphone_project` to show up on GitWeb, make the `repo` setting look like

```
[repo iphone_project]
daemon = yes
gitweb = yes
```

Now if you commit and push the project, GitWeb will automatically start showing the `iphone_project`.

4.10 Hosted Git

If you don't want to go through all of the work involved in setting up your own Git server, you have several options for hosting your Git projects on an external dedicated hosting site. Doing so offers a number of advantages: a hosting site is generally quick to set up and easy to start projects on, and no server maintenance or monitoring is involved. Even if you set up your own server internally, you may still want to use a public hosting site for your open source projects — it's generally easier for the open source community to collaborate.

These days you have a huge number of hosting options to choose from, each with different advantages and disadvantages. To see an up-to-date list, check out the following page:

```
https://git.wiki.kernel.org/index.php/GitHosting
```

Because I can't cover all of them, I'll use this section to walk through setting up an account and creating a new project at GitHub, which is where I work. This will give you an idea of what's involved.

GitHub is by far the largest open source Git hosting site and it's also one of the very few that offers both public and private hosting options so you can keep your open source and private commercial code in the same place. In fact, I used GitHub to privately collaborate on this book.

4.10.1 GitHub

GitHub is slightly different than most code-hosting sites in the way it uses namespaces in projects. Instead of being primarily based on the project, GitHub is user-centric. That means when I host my `grit` project on GitHub, you won't find it at `github.com/grit` but instead at `github.com/schacon/grit`. There's no standard name for any project, which allows a project to move from one user to another seamlessly if the first author abandons the project.

GitHub is also a commercial company that charges for accounts for maintaining private repositories, but anyone can quickly get a free account to host as many open source projects as they want. I'll quickly go over how that's done.

4.10.2 Setting Up a User Account

The first thing to do is to create a free user account. Visit the Pricing and Signup page at `http://github.com/plans` and click the "Create a free account" button on the right hand side of the page (see Figure 4-2). You're taken to the signup page.

Here you must choose a username that isn't yet taken and enter an e-mail address and password that will be associated with the account (see Figure 4-3).

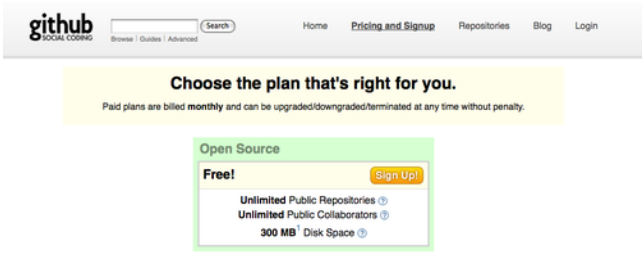

Figure 4.2: The GitHub plan page.

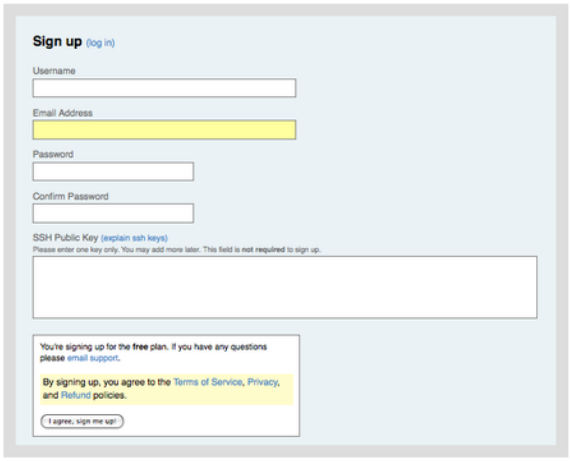

Figure 4.3: The GitHub user signup form.

If you have your public SSH key available, this is a good time to add it. I described how to generate a new SSH key pair earlier in the "Simple Setups" section. Take the contents of the public key of that pair, and paste it into the SSH Public Key text box. Clicking the "explain ssh keys" link takes you to detailed instructions on how to create SSH key pairs on all major operating systems. Clicking the "I agree, sign me up" button takes you to your new user dashboard (see Figure 4-4).

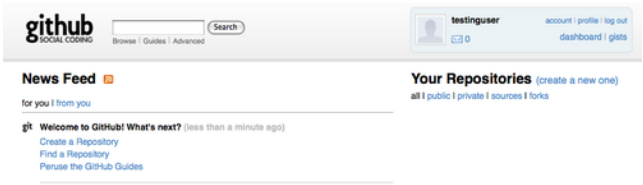

Figure 4.4: The GitHub user dashboard.

Next, create a new repository.

4.10.3 Creating a New Repository

Start by clicking the "create a new one" link next to "Your Repositories" on the user dashboard. You're taken to the "Create a New Repository" form (see Figure 4-5).

All you really have to do is provide a project name, but you can also add a description. When that's done, click the "Create Repository" button. Now you have a new repository on

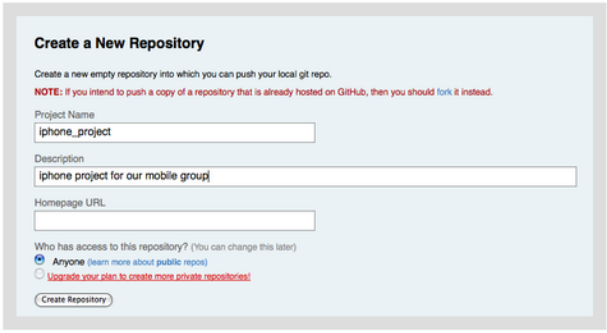

Figure 4.5: Creating a new repository on GitHub.

GitHub (see Figure 4-6).

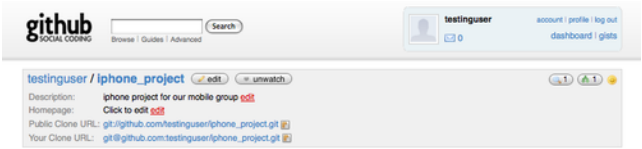

Figure 4.6: GitHub project header information.

Since you have no code there yet, GitHub will show instructions for how to create a brand-new Git project, push an existing project, or import a project from a public Subversion repository (see Figure 4-7).

Figure 4.7: Instructions for a new repository.

These instructions are similar to what I've already gone over. To initialize a project that isn't already a Git project, run

```
$ git init
$ git add .
$ git commit -m 'initial commit'
```

When you have a Git repository locally, add GitHub as a remote and push your master

branch.

```
$ git remote add origin git@github.com:testinguser/iphone_project.git
$ git push origin master
```

Now your project is hosted on GitHub, and you can give the URL to anyone you want to share your project with. In this case, it's `http://github.com/testinguser/iphone_project`. You can also see from the header on each of your project's pages that you have two Git URLs (see Figure 4-8).

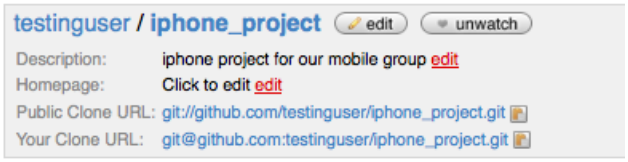

Figure 4.8: Project header with a public URL and a private URL.

The Public Clone URL is a public, read-only Git URL that anyone can use to clone the project. Feel free to give out that URL and post it wherever you want.

The Your Clone URL is a read/write SSH-based URL that you use to read or write only if you connect with the SSH private key associated with the public key you uploaded for your account. When other users visit this project page, they won't see that URL — only the public one.

4.10.4 Importing from Subversion

If you have an existing public Subversion project to import into Git, GitHub can often do most of the work for you. At the bottom of the instructions page is a link to Subversion import. If you click on it, you see a form with information about the import process and a text box where you paste the URL of your public Subversion project (see Figure 4-9).

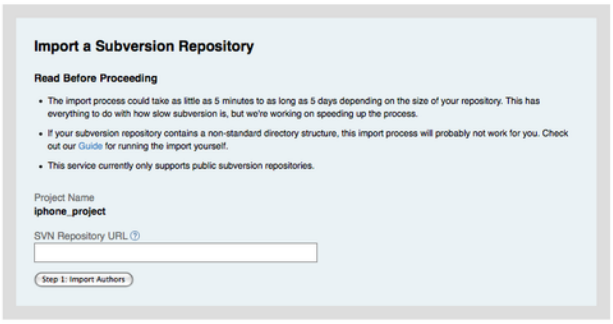

Figure 4.9: Subversion importing interface.

If your project is very large, nonstandard, or private, this process probably won't work. In Chapter 7, you'll learn how to manually import more complicated projects.

4.10.5 Adding Collaborators

Let's add the rest of the team. If John, Josie, and Jessica all sign up for accounts on GitHub, and you want to give them push access to your repository, add them to your project as collaborators. Doing so will allow pushes to your repository from accounts with their public keys to work.

Click the "edit" button in the project header or the "Admin" tab at the top of the project to reach the "Admin" page of your GitHub project (see Figure 4-10).

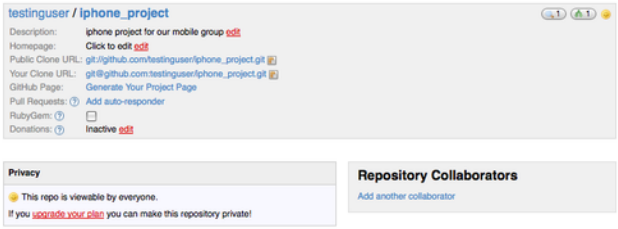

Figure 4.10: GitHub administration page.

To give another user write access to your project, click the "Add another collaborator" link. A new text box appears into which you type a GitHub username. As you type, a helper pops up, showing possible matches with other GitHub accounts. When you find the correct user, click the "Add" button to add that user as a collaborator to your project (see Figure 4-11).

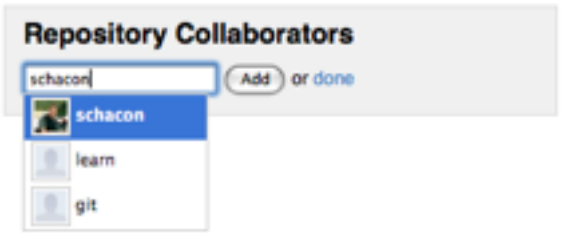

Figure 4.11: Adding a collaborator to your project.

When you're finished adding collaborators, you should see a collaborator list in the "Repository Collaborators" box (see Figure 4-12).

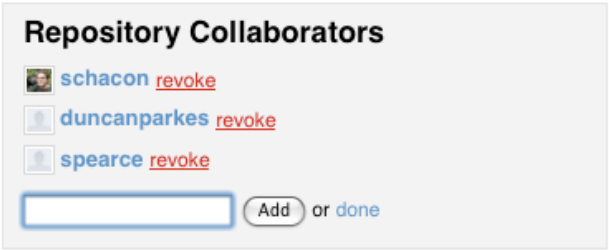

Figure 4.12: A list of collaborators on your project.

To revoke access to an individual, click the "revoke" link, and their push access will be removed. For future projects, you can also copy collaborator groups by copying the

permissions of an existing project.

4.10.6 Your Project

After you push your project or import it from Subversion, you have a main project page that looks something like Figure 4-13.

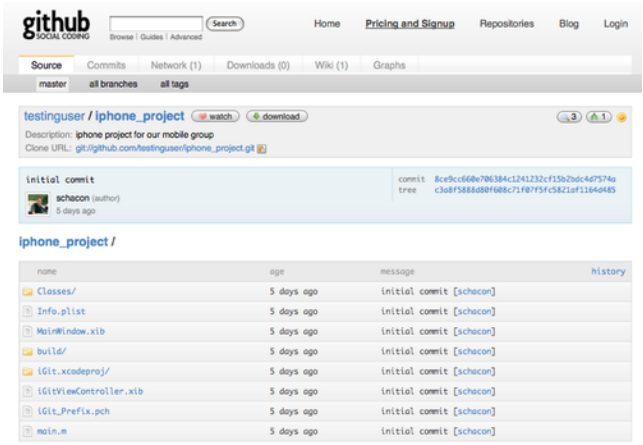

Figure 4.13: A GitHub main project page.

When people visit your project, they see this page. It contains tabs to different aspects of your projects. The “Commits” tab shows a list of commits in reverse chronological order, similar to the output of the `git log` command. The “Network” tab shows all the people who have forked your project and contributed back. The “Downloads” tab allows you to upload project binaries and link to tarballs and zipped versions of any tagged points in your project. The “Wiki” tab provides a wiki where you write documentation or other information about your project. The “Graphs” tab has contribution visualizations and statistics about your project. The main “Source” tab that you land on shows your project’s main directory listing and automatically renders the README file below it, if you have one. This tab also shows a box with the latest commit information.

4.10.7 Forking Projects

To contribute to an existing project to which you don’t have push access, GitHub encourages forking the project. When you land on a project page that looks interesting and you want to hack on it, click the “fork” button in the project header to have GitHub copy that project to your account so you can push to it after making changes.

This way, projects don’t have to worry about adding users as collaborators to give them write access. If people fork a project and push to it after making changes, then the project maintainer can pull those changes by adding the forks as remotes and merging the work they contain.

To fork a project, visit the project page (in this case, mojombo/chronic) and click the “fork” button in the header (see Figure 4-14).

After a few seconds, you’re taken to your new project page, which indicates that this project is a fork of another one (see Figure 4-15).

Figure 4.14: Get a writable copy of any repository by clicking the "fork" button.

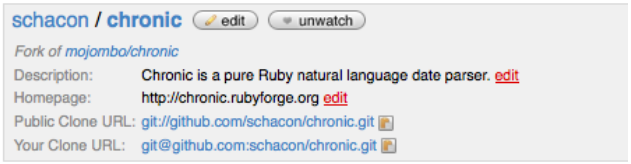

Figure 4.15: Your fork of a project.

4.10.8 GitHub Summary

That's all I'll mention about GitHub, but it's important to note how quickly you can do all this. You can create an account, add a new project, and push to it in a matter of minutes. If your project is open source, you also get a huge community of developers who now have visibility into your project and may well fork it and help contribute to it. At the very least, this may be a way to get up and running so you can try using Git to maintain it.

4.11 Summary

There are several ways to get a remote Git repository up and running to collaborate with others or share your work.

Running your own server gives you complete control and allows you to run the server within your own firewall. But, such a server generally requires a significant amount of time to set up and maintain. Placing your data on a hosted server is easy to set up and maintain. However, you have to be allowed to keep your code on someone else's server, which some organizations don't allow.

It should be fairly straightforward to determine which solution or combination of solutions is appropriate for you and your organization.

Chapter 5

Distributed Git

Now that you have a remote Git repository where developers can share their code, and you're familiar with running basic Git commands on a local repository, I'll cover how to use Git with remote repositories.

In this chapter, you'll see how to work with Git in a distributed environment as a contributor and as an integrator. You'll learn how to contribute code to a project in a way that makes it as easy on yourself and the project maintainer as possible. You'll also learn how to work with other developers to successfully maintain a project.

5.1 Distributed Workflows

Unlike CVSs, the distributed nature of Git allows far more flexibility in how developers collaborate on projects. In centralized systems, every developer interacts more or less equally in a central repository. In Git, however, every developer can both contribute code to other repositories and maintain a public repository on which others can base their changes and can contribute to. This opens a vast range of workflow possibilities for your project and/or your team, so I'll cover a few common patterns that take advantage of this flexibility. I'll go over the strengths and possible weaknesses of each design. Choose a single one to use, or mix and match features from each.

5.1.1 Centralized Workflow

In centralized systems, there's generally a single collaboration model — the centralized workflow. One central shared repository contains the official code, and all developers synchronize their changes to it (see Figure 5-1).

This means that if two developers checkout code from the central repository and both make changes, the first developer to check in their changes can do so without having to worry about merging. The second developer must merge in their changes without overwriting the first developer's changes.

If you have a small team or are already comfortable with a centralized workflow, you can 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. If one developer clones, makes changes, and then pushes their changes before another developer pushes conflicting changes, the Git server will reject the second developer's changes. They will be told that

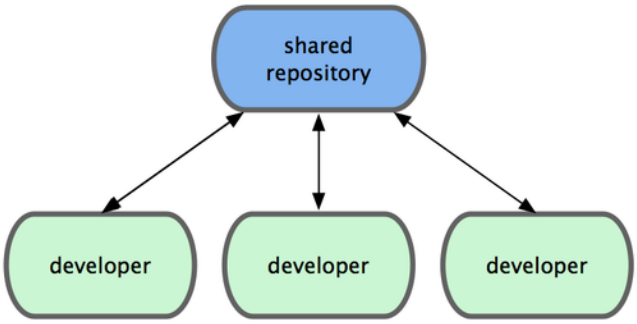

Figure 5.1: Centralized workflow.

they're trying to push non-fast-forward changes and that they won't be able to do so until they fetch and merge the first developers changes. This workflow is attractive because it's a style that many are familiar and comfortable with.

5.1.2 Integration-Manager Workflow

Because Git allows multiple remote repositories, it's possible to have a workflow where each developer has their own public repository that they push to, and read access to everyone else's. There's also a blessed repository containing the "official" project code. To contribute to that project, clone the blessed repository into a public repository and push your changes to it. Then, send a request to the integration manager to pull your changes. They add your repository as a remote, test your changes in their own repository, merge them into a branch in their repository, and then push to the blessed repository. The process works as follows (see Figure 5-2):

1. A developer clones the blessed repository into a private repository and makes changes there.
2. The developer pushes their changes to their public repository.
3. The developer sends the integration manager an e-mail message asking them to pull changes.
4. The integration manager adds the developer's public repository as a remote and merges it locally.
5. The integration manager pushes merged changes to the blessed repository.

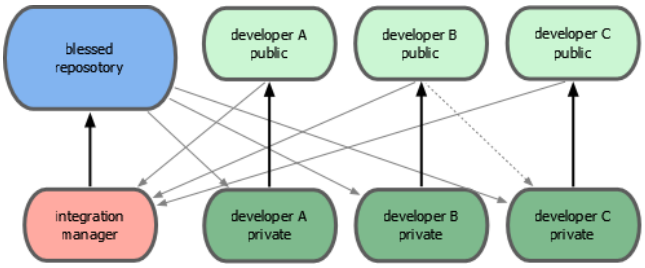

Figure 5.2: Integration-manager workflow.

This is a very common workflow with sites like GitHub, 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 in your local repository, and the maintainer of

the main repository can pull in your changes at any time. Contributors don't have to wait for the project maintainer to incorporate their changes. Each party can work at their own pace.

5.1.3 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. Above all the lieutenants in the chain of command is one integration manager known as the benevolent dictator. The benevolent dictator's repository serves as the reference repository from which all the collaborators need to pull. The process works like this (see Figure 5-3):

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

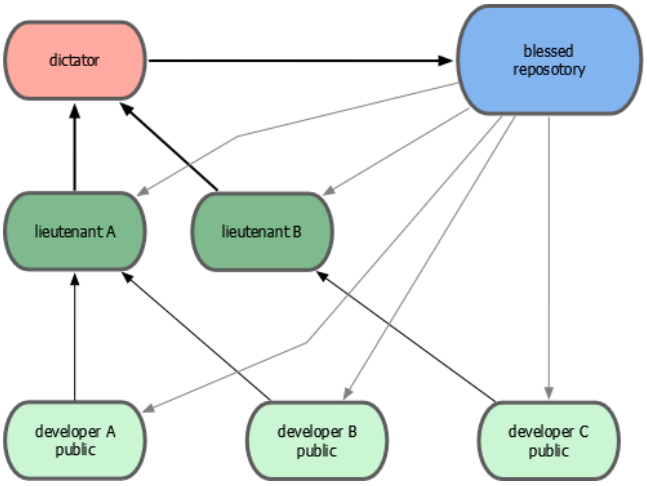

Figure 5.3: Benevolent dictator workflow.

This kind of workflow isn't common but can be useful in very big projects or in highly hierarchical environments, since 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.

These are some commonly used workflows that are possible with a distributed system like Git, but many other variations are possible to suit your particular real-world workflow. Now that you can (I hope) determine which workflow may work for you, I'll cover some more specific examples of how to accomplish the main roles that make up the different flows.

5.2 Contributing to a Project

You know what the different workflows are, and you should have a pretty good grasp of fundamental Git usage. In this section, you'll learn a few common patterns for contributing to a project.

The main difficulty with describing this process is the huge number of variations on how it's done. Because Git is very flexible, people can and do work together in many ways, and it's problematic to describe how you should contribute to a project. Every project is a bit different. Some of the variables involved are number of active contributors, chosen workflow, and commit access.

The first variable is the number of active contributors. How many users are actively contributing code to this project, and how often? In many instances, you'll have two or three developers making a few commits a day, or possibly fewer for dormant projects. In really large projects, the number of developers could be in the thousands, with dozens or even hundreds of patches coming in each day. The number is important because with more and more developers, you run into more issues with making sure changes apply cleanly or can be easily merged. Changes you submit may be rendered obsolete or severely broken by changes that are merged in while you were working or while your changes were waiting to be approved or applied. How can you keep your code consistently up to date and your patches valid?

The next variable is the workflow in use for the project. Is it centralized, with each developer having equal write access to the main codeline? Does the project have an integration manager who checks all the patches? Are all the patches peer-reviewed and approved? Are you involved in that process? Is a lieutenant system in place, and do you have to submit your changes to them first?

The next issue is your commit access. Contributing to a project is much different if you have write access to the project than if you don't. If you don't have write access, how does the project prefer to accept contributed work? Does it even have a policy? How much work are you contributing at a time? How often do you contribute?

All these questions can affect how you contribute effectively to a project and what workflows are preferred or available. I'll cover aspects of each of these in a series of use cases, moving from simple to more complex. You should be able to construct the specific workflows you need in practice from these examples.

5.2.1 Commit Guidelines

Before you start looking at specific use cases, here's a quick note about commit messages. Having good guideline sfor creating commits and sticking to them 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 — it's in the Git source code in the `Documentation/SubmittingPatches` file.

First, don't include any whitespace errors in your commit. Git provides an easy way to check for this. Before you commit, run `git diff --check`, which lists possible whitespace errors. Here's an example, where I've replaced the red terminal color with `X`s.

```
$ git diff --check
lib/simplegit.rb:5: trailing whitespace.
+    @git_dir = File.expand_path(git_dir)XX
lib/simplegit.rb:7: trailing whitespace.
+ XXXXXXXXXXX
```

```
lib/simplegit.rb:26: trailing whitespace.
+    def command(git_cmd)XXXX
```

If you run this before committing, you'll see if your commit contains whitespace issues that may annoy other developers.

Next, try to make each commit logically self-contained. If you can, try to make your changes digestible. Don't spend a whole weekend working on five different issues and then submit all your changes as one massive commit on Monday. Even if you don't commit during the weekend, use the staging area on Monday to split your changes 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 Chapter 6). The project snapshot at the tip of the branch is identical whether you make 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 a commit later. Chapter 6 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.

The last thing to keep in mind is the commit message. Getting in the habit of creating quality commit messages makes collaborating with other developers a lot easier. As a general rule, your messages should start with a single line that's no more than about 50 characters long that describes the commit 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. It's also a good idea to use the imperative present tense in these messages. In other words, use commands. Instead of "I added tests for" or "Adding tests for," use "Add tests for." Here's a template originally written by Tim Pope at tpope.net:

```
Short (50 chars or less) summary of changes

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 can get confused if you run the
two together.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded by a
   single space, with blank lines in between, but conventions vary here
```

If all your commit messages look like this, things will be a lot easier for you and the developers you work with. The Git project has well-formatted commit messages. I encourage you to run `git log --no-merges` in your copy of the Git repository to see what a nicely formatted project-commit history looks like.

In the following examples, and throughout most of this book, for the sake of brevity I don't format messages nicely like this. Instead, I use the `-m` option to `git commit`. Do as I say, not as I do.

5.2.2 Private Small Team

The simplest setup you're likely to encounter is a private project with one or two other developers. By private, I mean closed source — not visible to the outside world. You and the other developers all have push access to the repository.

In this environment, you 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 on the clients 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. (I'm replacing the protocol messages with `...` in these examples to shorten them.)

```
# John's Machine
$ git clone john@githost:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 file changed, 1 insertion(+), 1 deletion(-)
```

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

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 file changed, 1 insertion(+), 0 deletions(-)
```

Now, Jessica pushes her changes up to the server.

```
# Jessica's Machine
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

John tries to push his change up, too.

```
# 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'
```

John isn't allowed to push because Jessica has pushed in the meantime. 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 a merge on the server if different files are edited, in Git you must merge the commits locally. John has to fetch Jessica's changes and merge them in before he will be allowed to push.

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

At this point, John's local repository looks something like Figure 5-4.

John has a reference to the changes Jessica pushed, but he has to merge them into his own changes before he's allowed to push.

```
$ git merge origin/master
Merge made by recursive.
 TODO |    1 +
 1 file changed, 1 insertion(+), 0 deletions(-)
```

The merge goes smoothly. John's commit history now looks like Figure 5-5.

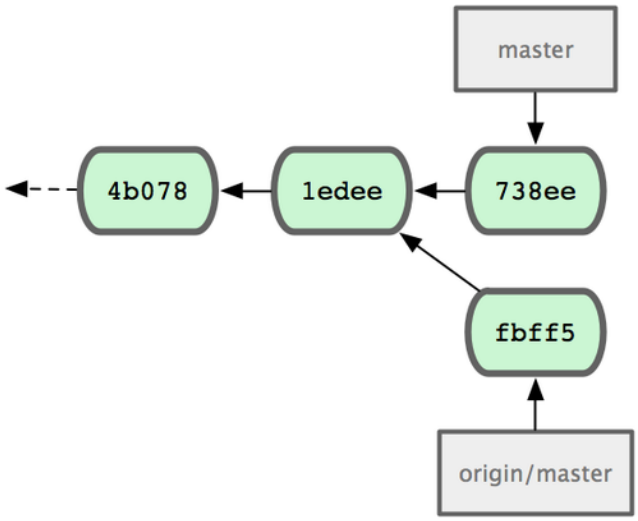

Figure 5.4: John's initial repository.

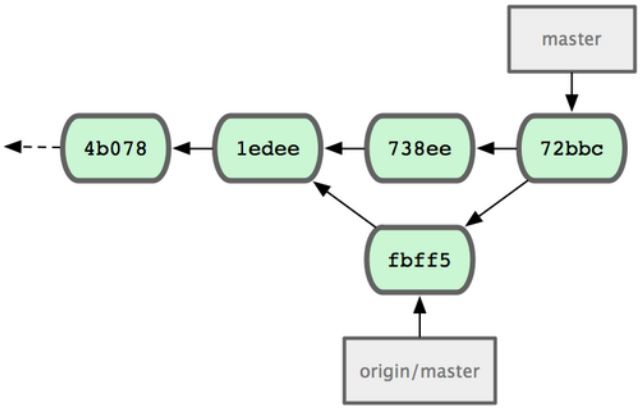

Figure 5.5: John's repository after merging origin/master.

Now, John can test his code to make sure it still works properly, and then he can push his new merged changes to the server.

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

Finally, John's commit history looks like Figure 5-6.

In the meantime, Jessica has been working on a topic branch. She's created a topic branch called `issue54` and made three commits on that branch. She hasn't fetched John's changes yet, so her commit history looks like Figure 5-7.

Jessica wants to sync up with John, so she fetches.

```
# Jessica's Machine
$ git fetch origin
```

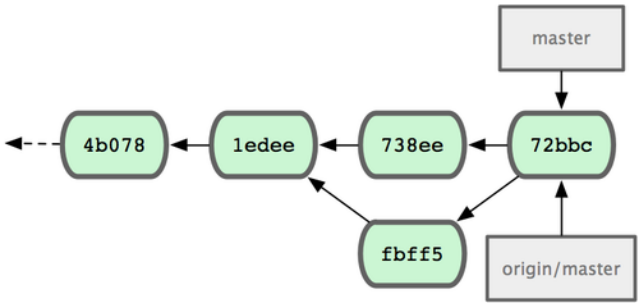

Figure 5.6: John's history after pushing to the origin server.

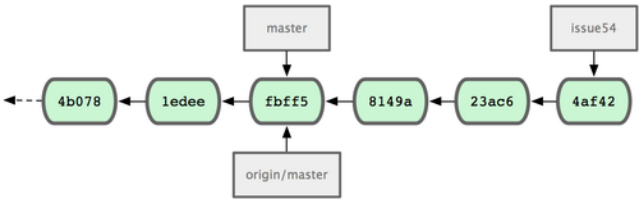

Figure 5.7: Jessica's initial commit history.

```
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

That pulls down the changes John has pushed in the meantime. Jessica's history now looks like Figure 5-8.

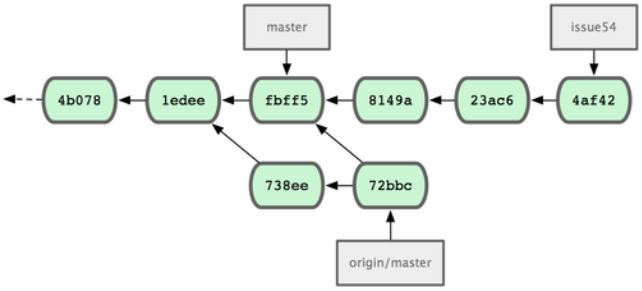

Figure 5.8: Jessica's history after fetching John's changes.

Jessica thinks her topic branch is ready, but she wants to know where she has to merge her changes so that she can push. She runs `git log` to find out.

```
$ git log --no-merges origin/master ^issue54
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

    removed invalid default value
```

Now, Jessica can merge her topic branch into her master branch, merge John's changes (`origin/master`) into her `master` branch, and then push to the server again. First, she switches back to her master branch to integrate all this work.

```
$ git checkout master
Switched to branch "master"
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

She can merge either `origin/master` or `issue54` first. They're both upstream, so the order doesn't matter. The end result will be identical no matter which order she chooses. Only the history will be different. She chooses to merge in `issue54` first.

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletion(-)
```

No problems occur. As you can see, it was a simple fast-forward. Now Jessica merges in John's changes (`origin/master`).

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Everything merges cleanly, and Jessica's history looks like Figure 5-9.

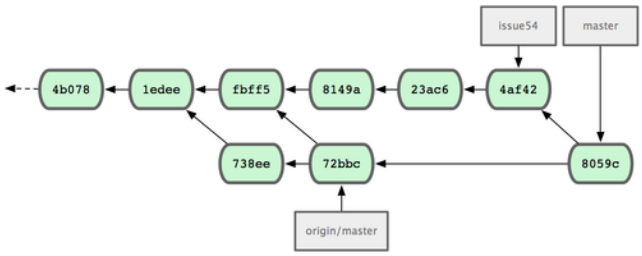

Figure 5.9: Jessica's history after merging John's changes.

Now `origin/master` is reachable from Jessica's `master` branch, so she should be able to successfully push (assuming John hasn't pushed again 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 changes successfully (see Figure 5-10).

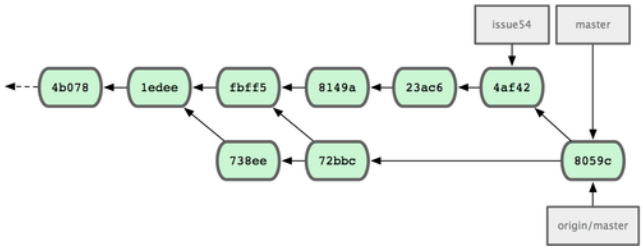

Figure 5.10: Jessica's history after pushing all changes back to the server.

That's one of the simplest workflows. Work for a while, generally in a topic branch, and merge into your master branch when the topic branch is ready to be integrated. When you want to share that work, fetch and merge `origin/master` if it has changed, and finally push to the `master` branch on the server. The general sequence is something like that shown in Figure 5-11.

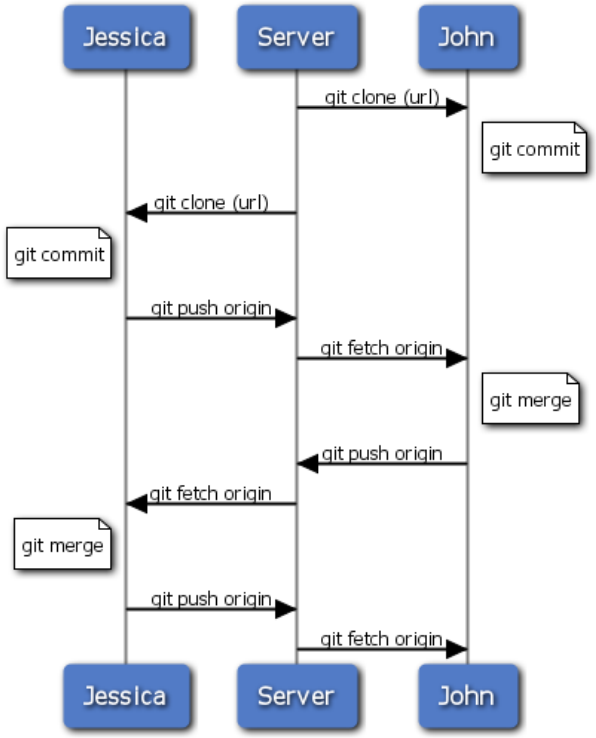

Figure 5.11: General sequence of events for a simple multiple-developer Git workflow.

5.2.3 Private Managed Team

This next scenario looks at contributor roles in a larger private group. You'll learn how to work in an environment where small groups collaborate on features and then those team-based contributions are integrated by somebody else.

Let's say that John and Jessica are working together on one feature, while Jessica and Josie are working on a second. In this case, the company is using a type of integration-manager workflow where the changes of the individual groups are integrated only by certain engineers, and the `master` branch of the main repository can be updated only by those engineers. In this scenario, all changes are 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. Assuming she already clones her repository, 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 file changed, 1 insertion(+), 1 deletion(-)
```

At this point, she needs to share her changes with John, so she pushes her `featureA` branch commits to the Git server. Jessica doesn't have push access to the `master` branch — only the integrators do — so she has to push to `featureA` in order to collaborate with John.

```
$ git push origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica e-mails John to tell him that she's pushed some changes into a branch named `featureA` on the Git server 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, `featureB`, 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 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
 1 file changed, 1 insertion(+), 1 deletion(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Jessica's repository looks like Figure 5-12.

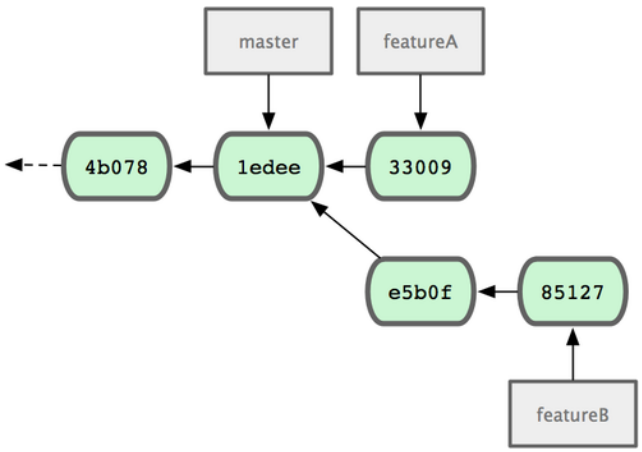

Figure 5.12: Jessica's initial commit history.

Jessica's ready to push her work, but she gets an e-mail from Josie saying that she (Josie) created a branch with some initial changes in it called `featureBee` that she already pushed to the Git server. Jessica first needs to merge those changes into her `featureB` branch before she can push it to the Git server. She then fetches Josie's changes by running `git fetch`.

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Jessica can now merge this into the changes she did with `git merge`.

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    4 ++++
 1 file changed, 4 insertions(+), 0 deletions(-)
```

There's a bit of a problem here. She needs to push the merged changes in her `featureB` branch to the `featureBee` branch on the server. She can do so by specifying the local branch followed by a colon (:), followed by the remote branch to the `git push` command.

```
$ git push origin featureB:featureBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  featureB -> featureBee
```

This is called a *refspec*. See Chapter 9 for a more detailed discussion of Git refspecs.

Next, John e-mails Jessica to say he's pushed some changes to the `featureA` branch and asks her to verify them. She runs `git fetch` to pull those changes.

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  featureA   -> origin/featureA
```

She can then see what has changed by running `git log`.

```
$ git log origin/featureA ^featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    changed log output to 30 from 25
```

Finally, she merges John's changes into her own `featureA` branch.

```
$ git checkout featureA
Switched to branch "featureA"
$ git merge origin/featureA
```

```
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb |   10 +++++++++-
1 file changed, 9 insertions(+), 1 deletion(-)
```

Jessica wants to tweak something, so she commits again and then pushes this commit back to the server.

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push origin featureA
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

Jessica's commit history now looks something like Figure 5-13.

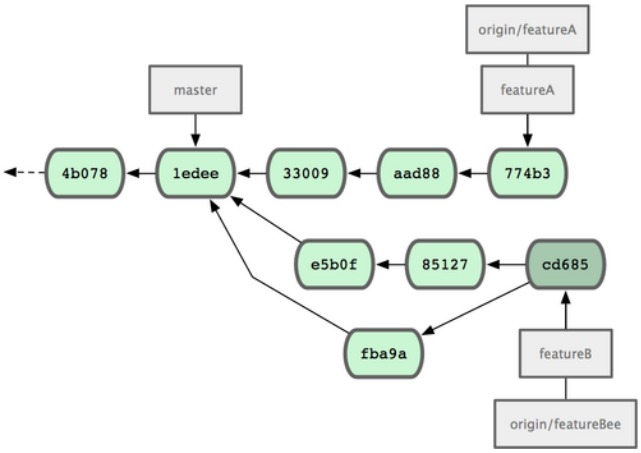

Figure 5.13: Jessica's history after committing on a feature branch.

Jessica, Josie, and John inform the integrator that the `featureA` and `featureBee` branches on the server are ready for integration into the mainline. After the integrator incorporates these branches into the mainline, when Jessica runs `git fetch` this brings down the new merge commits into her repository, making her repository look like Figure 5-14.

Many groups switch to Git because of this ability to have multiple teams working in parallel, merging different branches 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 Figure 5-15.

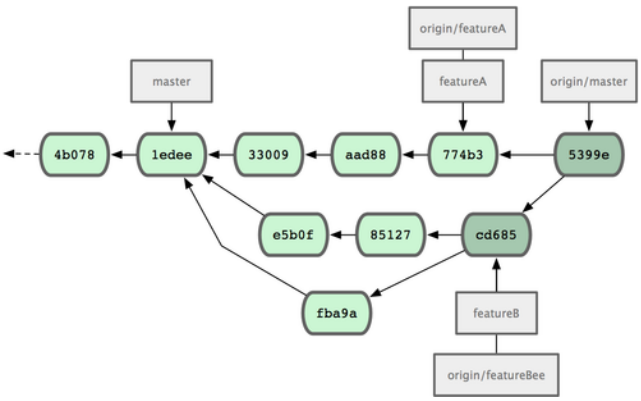

Figure 5.14: Jessica's repository after merging both her topic branches.

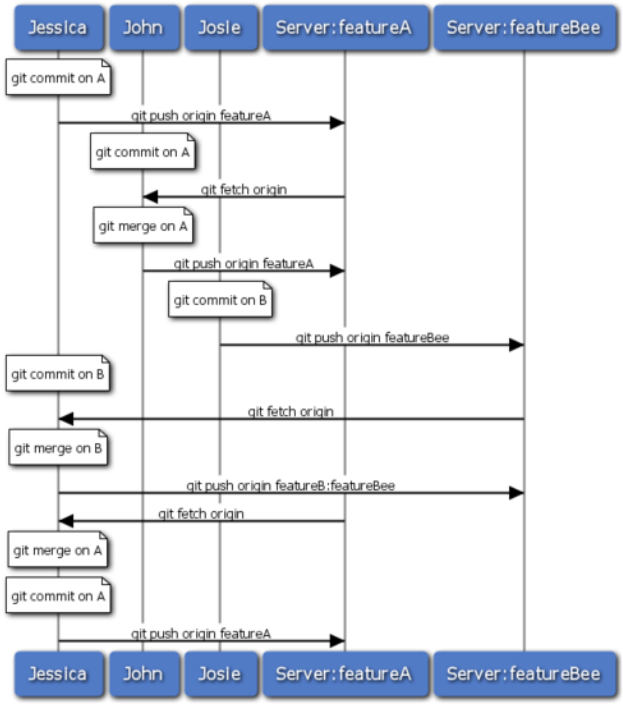

Figure 5.15: Basic sequence of this managed-team workflow.

5.2.4 Public Small Project

Contributing to public projects is a bit different. Because you don't have permission to directly update branches in the project, you have to supply your changes to the maintainers some other way. This first example describes contributing via forking on Git hosting sites that support easy forking. The repo.or.cz and GitHub hosting sites both support this, and many project maintainers expect this style of contribution. The next section deals with projects that prefer to accept contributed patches via e-mail.

Start by cloning the main repository, creating a topic branch for the patch you're planning to contribute, and start doing your work there. This looks basically like

```
$ git clone (url)
$ cd project
```

```
$ git checkout -b featureA
$ (work)
$ git commit
$ (work)
$ git commit
```

You may want to run `git rebase -i` to squash your changes down to a single commit, or rearrange the changes in the commits to make the patch easier for the maintainer to review. See Chapter 6 for more information about interactive rebasing.

When you're finished working in your topic branch and you're ready to contribute it back to the maintainer, go to the original project page and click the "Fork" button, creating your own writable fork of the project. Add this new repository URL as a second remote, in this case named `myfork`.

```
$ git remote add myfork (url)
```

Push your changes to the fork. It's easiest to push the branch you're working on to your fork, rather than merging into your master branch and pushing that. The reason is that if the changes aren't accepted or are cherry picked, you don't have to rewind your master branch. If the maintainers merge, rebase, or cherry-pick your work, you'll eventually get it back by pulling from their repository anyhow.

```
$ git push myfork featureA
```

After you push your changes to your fork, notify the maintainer. This is often called a pull request, and either generate it via the website — GitHub has a "pull request" button that automatically sends a message the maintainer — or run the `git request-pull` command and e-mail the 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 to pull from, and outputs 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 runs

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
        added a new function

are available in the git repository at:
```

```
  git://githost/simplegit.git featureA

Jessica Smith (2):
      add limit to log function
      change log output to 30 from 25

 lib/simplegit.rb |   10 +++++++++-
 1 file changed, 9 insertions(+), 1 deletion(-)
```

She sends the output to the maintainer. It tells them where the changes were branched from, summarizes the commits, and tells where to pull from.

On a project for which you're not the maintainer, it's generally easier to have your `master` branch always track `origin/master` and to do your work in topic branches that you can easily discard if your changes are rejected. Isolating work into topic branches also makes it easier for you to rebase your changes if the tip of the main repository has moved in the meantime and your commits no longer apply cleanly. For example, to submit a second topic branch to the project, don't continue working on the topic branch you just pushed. Start over from the main repository's `master` branch.

```
$ git checkout -b featureB origin/master
$ (work)
$ git commit
$ git push myfork featureB
$ (email 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. Figure 5-16 shows this.

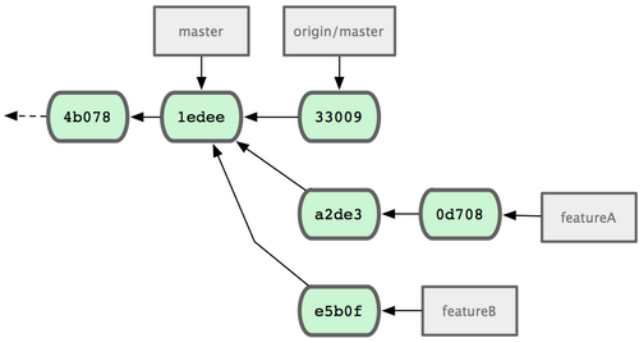

Figure 5.16: Initial commit history with featureB work.

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, 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 Figure 5-17.

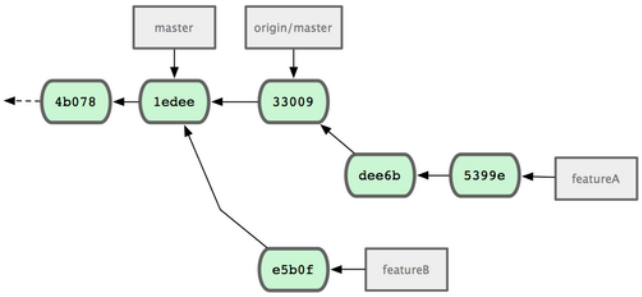

Figure 5.17: Commit history after featureA work.

Because you rebased the branch, you have to specify the `-f` option on your push command in order 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 branch to a different branch on the server (perhaps called `featureAv2`).

Let's look at one more possible scenario: the maintainer has looked at the changes 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 changes 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 --no-commit --squash featureB
$ (change implementation)
$ git commit
$ git push myfork featureBv2
```

The `--squash` option takes all the changes on the merged branch and squashes them into one non-merge commit on top of the branch you're on. The `--no-commit` option tells Git not to automatically record a commit. This allows you to incorporate all the changes from another branch and then make more changes before recording the new commit.

Now send the maintainer a message saying that you've made the requested changes and they can find those changes in your `featureBv2` branch (see Figure 5-18).

5.2.5 Public Large Project

Many larger projects have established procedures for accepting patches. You'll need to check the specific rules for each project because they will differ. However, many larger

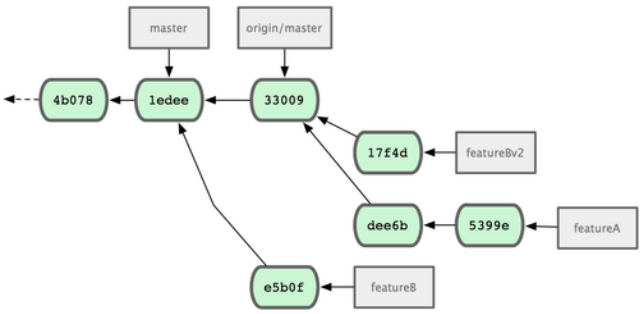

Figure 5.18: Commit history after featureBv2 work.

public projects accept patches via a developer mailing list, so I'll go over an example of that now.

The workflow is similar to the previous use case. Create topic branches for each patch series you work on. The difference is how you submit the patches to the project. Instead of forking the project and pushing to your own writable version, generate e-mail versions of each patch series and e-mail 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. Use `git format-patch` to generate the mbox-formatted files that you e-mail to the list. It turns each commit into an e-mail message with the first line of the commit message as the subject and the rest of the commit message, plus the patch that the commit introduces, as the body. The nice thing about this is that applying a patch from an e-mail generated with `git format-patch` preserves all the commit information properly, as you'll see in the next section when you apply these patches.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

The `git format-patch` command outputs the names of the patch files it creates. The `-M` switch tells Git to look for renames. The files end up looking like

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

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')
--
1.6.2.rc1.20.g8c5b.dirty
```

You can also edit these patch files to add more information for the e-mail 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 `lib/simplegit.rb` line), then developers can read it in the email message; but applying the patch excludes it.

To e-mail this to an email list, either paste the file into your e-mail program or send it via a command-line email 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. I'll demonstrate how to send a patch via Gmail, which happens to be the e-mail client many developers use. Read detailed instructions for a number of mail programs at the end of the aforementioned `Documentation/SubmittingPatches` file in the Git source code.

First, set up the imap section in your `~/.gitconfig` file. Set each value separately with a series of `git config` commands, or add them manually by editing the file with your favorite text editor. But in the end, your config file should look something like

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
```

```
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

If your IMAP server doesn't use SSL, the last two lines aren't necessary, and the host value will start with `imap://` instead of `imaps://`. When your config file is set up, use `git send-email` to place the patch series in the Drafts folder of the specified IMAP server.

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-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 produces a bunch of log information that looks 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] added 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
```

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

5.2.6 Summary

This section covered a number of common workflows for dealing with several very different types of Git project collaboration styles you're likely to encounter and introduced a couple of new tools to help you manage this process. Next, you'll see how to work the other side of the coin: maintaining a Git project. You'll learn how to be a benevolent dictator or integration manager.

5.3 Maintaining a Project

In addition to knowing how to effectively contribute to a project, you'll likely need to know how to maintain one. This can consist of accepting and applying patches generated via `git format-patch` and e-mailed to you, or integrating changes in remote branches for repositories you've added as remotes to your project. Whether you maintain a canonical repository or want to help by verifying or approving patches, you need to know how to accept changes in a way that's clearest to other contributors and sustainable by you over the long run.

5.3.1 Working in Topic Branches

When you integrate new changes, it's generally a good idea to try doing it in a topic branch — a temporary branch specifically made to try out that new idea. This way, it's easy to work on a patch separately and switch to another branch until you have time to come back to it if the patch isn't working. If you create a simple branch name based on the theme of the change, such as `ruby_client` or something similarly descriptive, you can easily remember what the branch is for. 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 changes. As I mentioned, create a branch based off your master branch like this.

```
$ git branch sc/ruby_client master
```

Or, to also switch to it immediately, use the `checkout -b` command.

```
$ git checkout -b sc/ruby_client master
```

Now you're ready to add your changes into this topic branch and determine if you want to merge it into branches that you're planning on keeping for a while.

5.3.2 Applying Patches from E-mail

If you receive a patch over e-mail to integrate into your project, apply the patch in a topic branch to evaluate it. There are two ways to apply an e-mailed 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 the Unix `diff` command, apply the patch with `git apply`. Assuming you saved the patch in `/tmp/patch-ruby-client.patch`, apply the patch like this.

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

This modifies files in your working directory. It's almost identical to running a `patch -p1` command to apply the patch, although `git apply` 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` follows 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 paranoid than `patch`. It won't create a commit for you. After running it, you must manually stage and commit the changes.

You can also run `git apply --check` to see if a patch applies cleanly before you try actually applying it.

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

If there's 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.

Applying a Patch with am

If the contributor is a Git user and was good enough to use the `git format-patch` command to generate their patch, then your job is easier because the patch contains author information and a commit message. If you can, encourage your contributors to use `git format-patch` instead of `diff` to generate patches.

To apply a patch generated by `git format-patch`, use `git am`. Technically, `git am` is built to read an mbox file, which is a simple, plain-text format for storing one or more e-mail messages in one text file. Such a file might look something like this:

```
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
```

This is the beginning of the output of the `git format-patch` command that you saw in the previous section. This is also in valid mbox e-mail format. If someone has e-mailed you the patch properly using `git send-email`, and you save the patch in mbox format, then point `git am` to that mbox file, and `git am` will start applying all the patches it sees. If you run a mail client that can save several e-mails in mbox format, you can save an entire patch series in one file and then use `git am` to apply the patches one at a time.

However, if someone uploaded a patch file generated via `git format-patch` to a ticketing system or something similar, save the file locally and then pass its filename to `git am` to apply it.

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

This shows that the patch went in cleanly and `git am` automatically created a new commit. The author information is taken from the e-mail's `From` and `Date` headers, and the message of the commit is taken from the `Subject` and body (before the patch) of the e-mail. For example, if the patch from the mbox example showed above were applied, the resulting commit would look something like

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

   add limit to log function

   Limit log functionality to the first 20
```

The `Commit` information indicates the person who applied the patch and the time it was applied. The `Author` information is the individual who originally created the patch and when it was created.

But it's possible that the patch doesn't apply cleanly. Perhaps your main branch has diverged too far from the branch the patch was built from, or the patch depends on another patch you haven't applied yet. In that case, `git am` will fail and ask what you want to do.

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

```
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

This command puts conflict markers in any files it has issues with, much like a conflicted merge or rebase operation. Solve this issue much the same way. Edit the file to resolve the conflict, stage the new file, and then run `git am --resolved` to continue to the next patch.

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

For Git to try a bit more intelligently to resolve the conflict, include the `-3` option, which makes Git attempt a three-way merge. This option isn't on by default because it doesn't work if the commit the patch says it was based on isn't in your repository. If you do have that commit — if the patch was based on a public commit — then the `-3` option is generally much smarter about resolving a conflicting patch.

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

In this case, I was trying to apply a patch I'd already applied. Without the `-3` option, it looks like a conflict.

If you're applying a number of patches from an mbox file, you can also run `git am` in interactive mode, which stops at each patch it finds and asks if you want to apply it.

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
seeing if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

This is nice if you have a number of saved patches, because you can view the patch first if you don't remember what it is, or choose not apply the patch if it's already been applied.

When all the patches for your topic are applied and committed into your branch, choose whether and how to integrate them into a longer-lived branch.

5.3.3 Checking Out Remote Branches

If you received a contribution from a user who set up their own Git repository, pushed a number of changes into it, and then sent you the URL to the repository and the name of the remote branch the changes are in, add the repository as a remote and merge locally.

For instance, if Jessica sends you an e-mail saying that she has a great new feature in the `ruby-client` branch of her repository, test it by adding the remote branch and checking it out locally.

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

If she e-mails you again later with another branch containing another great feature, you can fetch and checkout the branch because you already have the remote set up.

This is most useful if you're working with a person on a regular basis. If someone only has a single patch to contribute once in a while, then accepting it over e-mail may be less time consuming than requiring everyone to run their own Git server and you having to continually add and remove remotes to get a few patches. You're also unlikely to want hundreds of remotes, each for someone who contributes only a patch or two. However, scripts and hosted services may make this easier. It depends largely on how you and your contributors collaborate.

The other advantage of this approach is that you get the history of the commits as well. Although you may have legitimate merge issues, you know where in your history their changes are based. A proper three-way merge is the default rather than having to supply the `-3` option and hope the patch was generated off a public commit to which you have access.

If you aren't working with a person consistently but still want to pull from them in this way, provide the URL of the remote repository to the `git pull` command. This does a one-time pull and doesn't save the URL as a remote reference.

```
$ git pull git://github.com/onetimeguy/project.git
From git://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by recursive.
```

5.3.4 Determining What Is Introduced

Now you have a topic branch that contains contributed changes. At this point, determine what you'd like to do with it. This section revisits a couple of commands to show how to review exactly what you'll be introducing if you merge the changes into your main branch.

It's often helpful to review all the commits in this branch but not in your master branch. Exclude commits in the master branch by adding the `--not` option before the branch name. For example, if your contributor sends two patches and you create a branch called `contrib` where you apply those patches, run

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

To see what changes each commit introduces, remember to pass the `-p` option to `git log` to append the diff introduced in each commit.

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

```
$ git diff master
```

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

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

What you really want to see are the changes added to the topic branch — the changes you'll introduce if you merge this branch with master. Do that by comparing the last commit on your topic branch with the first common ancestor it has with the master branch.

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

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

However, that isn't convenient, so Git provides a shorthand for doing the same thing: the triple-dot syntax. In a `git diff` command, put three periods after another branch to do a `diff` between the last commit of the branch you're on and its common ancestor with the other branch.

```
$ git diff master...contrib
```

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

5.3.5 Integrating Contributed Changes

When all the changes in your topic branch are ready to be integrated into a more mainline branch, the question is how to do it. Furthermore, what overall workflow do you want to use to maintain your project? You have a number of choices, so I'll cover a few of them.

Merging Workflows

One simple workflow merges your changes into your `master` branch. In this scenario, you have a `master` branch that contains basically stable code. When you've verified the changes in a topic branch that you created or that someone else has contributed, merge the topic branch into your master branch, delete the topic branch, and then continue the process. If you have a repository with changes in two branches named `ruby_client` and `php_client` that looks like Figure 5-19, and you merge `ruby_client` first and then `php_client` next, then your history will end up looking like Figure 5-20.

That's probably the simplest workflow, but it's problematic if you're dealing with larger repositories or projects.

If you have more developers or a larger project, you'll probably want to use at least a two-phase merge cycle. In this scenario, you have two long-running branches, `master` and `develop`. `master` is updated only when a very stable release is ready and all new code is integrated into the `develop` branch. You regularly push both of these branches to the public repository. Each time you have a new topic branch to merge in (Figure 5-21), merge it into `develop` (Figure 5-22). Then, when you tag a release, fast-forward `master` to wherever the now-stable `develop` branch is (Figure 5-23).

This way, when somebody clones your project's repository, they can either checkout `master` to build the latest stable version and keep `master` up to date, or they can check out

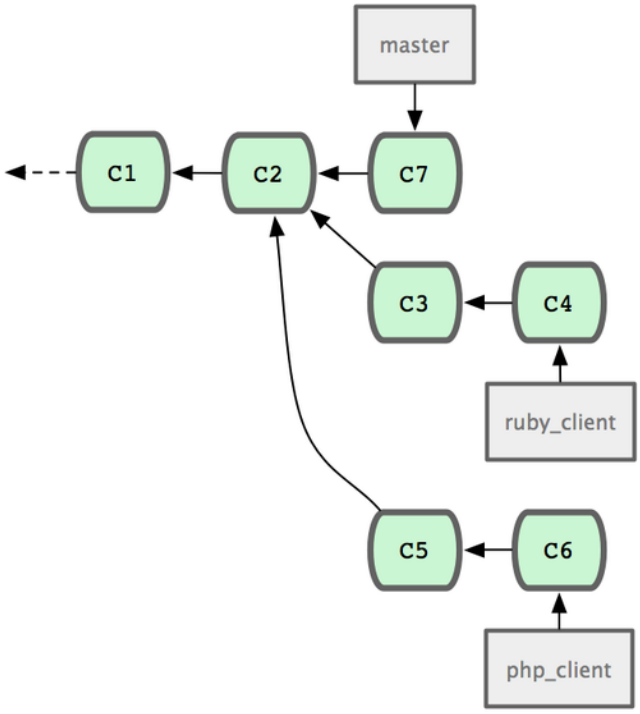

Figure 5.19: History with several topic branches.

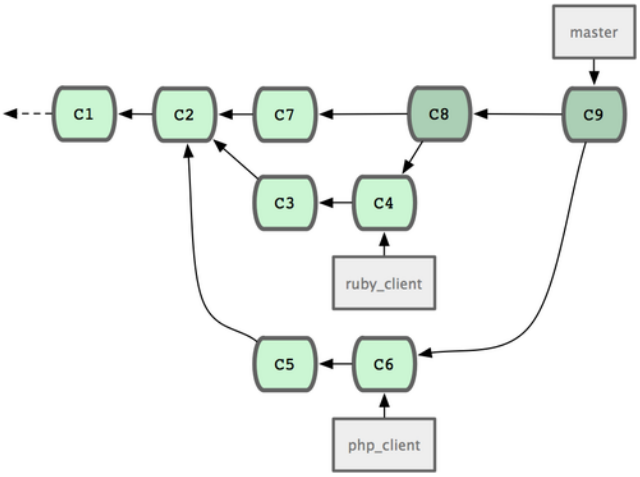

Figure 5.20: After a topic branch merge.

`develop`, which is the more cutting-edge stuff. You can also continue this concept, having a branch for integration where all the changes are merged together. Then, when the codebase on that branch is stable and passes any tests, merge it into a `develop` branch. When that has proven itself stable for a while, fast-forward your `master` branch.

Large-Merging Workflows

The Git project has four long-running branches: `master`, `next`, `pu` (proposed updates) for new work, and `maint` for maintenance backports. When new changes are introduced by contributors, they're collected into topic branches in the maintainer's repository in a manner similar to what I've described (see Figure 5-24). At this point, the maintainer evaluates the topics to determine whether they're safe and ready for consumption or whether they need

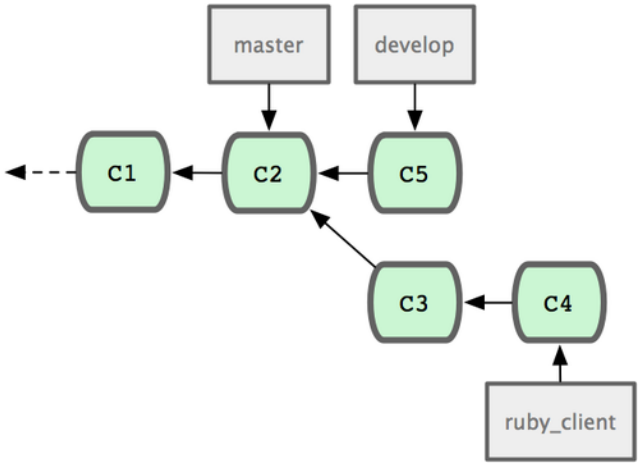

Figure 5.21: Before a topic branch merge.

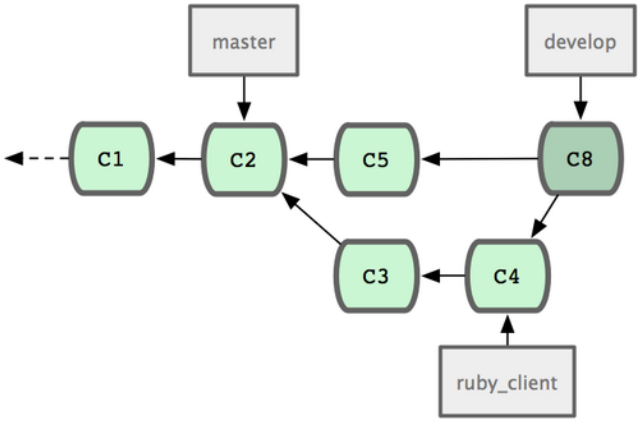

Figure 5.22: After a topic branch merge.

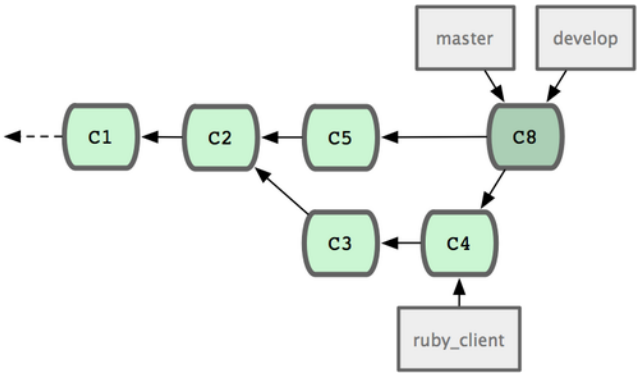

Figure 5.23: After a topic branch release.

more work. If they're safe, they're merged into `next`, and that branch is pushed so everyone can try the topics integrated together.

If the topic branches still need work, they're merged into `pu` instead. When it's determined that they're totally stable, the topics are re-merged into `master` and are then rebuilt from the topic branches that were in `next` but didn't yet graduate to `master`. This means `master` almost always moves forward, `next` is rebased occasionally, and `pu` is rebased even more often (see Figure 5-25).

When a topic branch has finally been merged into `master`, it's removed from the reposi-

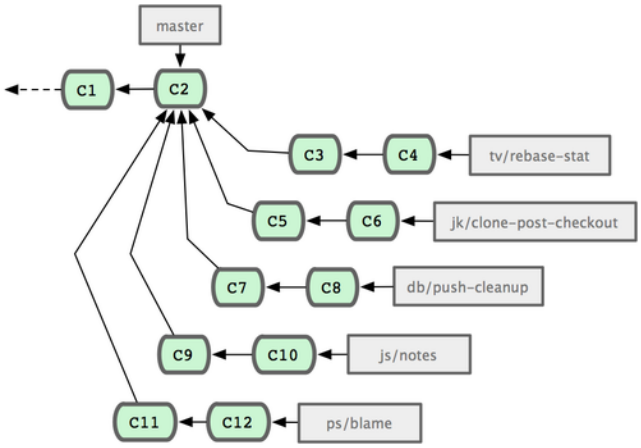

Figure 5.24: Managing a complex series of parallel contributed topic branches.

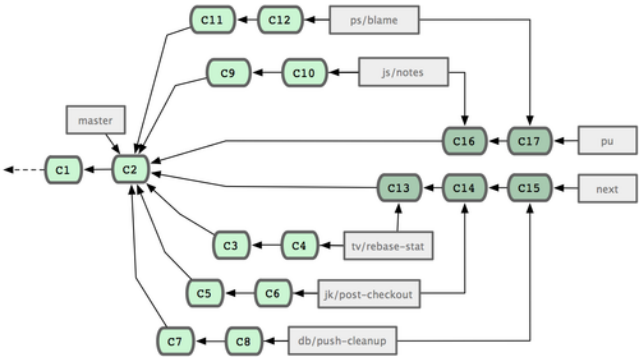

Figure 5.25: Merging contributed topic branches into long-term integration branches.

tory. The Git project also has a `maint` branch that's forked off from the last release to provide backported patches in case a maintenance release is required. Thus, when you clone the Git repository, you have four branches that you can check out to evaluate the project in different stages of development, depending on how cutting edge you want to be or how you want to contribute. The maintainer has a structured workflow to help vet new contributions.

Rebasing and Cherry Picking Workflows

Other maintainers prefer to rebase or cherry-pick contributed changes on top of their master branch, rather than merging them, to keep a mostly linear history. When you have changes in a topic branch and have determined that you want to integrate them, move to that branch and run `git rebase` to rebuild the changes on top of your current `master` (or `develop`, and so on) branch. If that works well, fast-forward your `master` branch, and you'll end up with a linear project history.

The other way to move introduced changes from one branch to another is to cherry-pick them. A cherry-pick in Git is like a rebase for a single commit. It takes the patch that was introduced in a commit and tries to reapply it on the branch you're currently on. This is useful if you have a number of commits on a topic branch and you want to integrate only one of them, or if you only have one commit on a topic branch and you'd prefer to cherry-pick it rather than run `git rebase`. For example, suppose you have a project that looks like Figure 5-26.

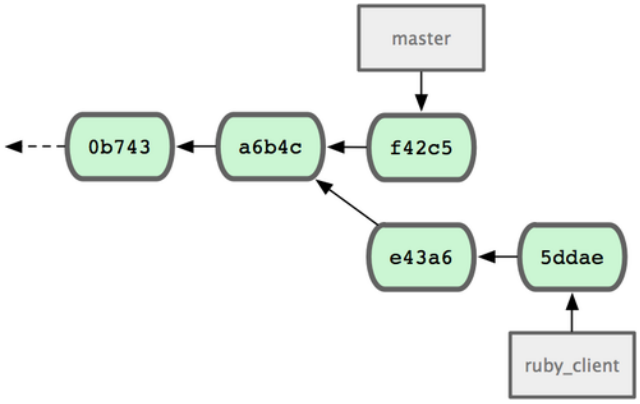

Figure 5.26: Example history before a cherry pick.

To pull commit `e43a6` onto your master branch, run

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

This pulls the same change introduced in `e43a6`, but you get a new commit SHA-1 hash, because the date when you did the commit is different. Now your history looks like Figure 5-27.

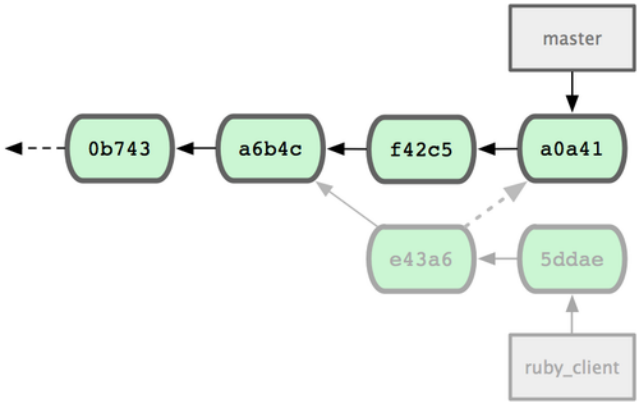

Figure 5.27: History after cherry-picking a commit on a topic branch.

Now remove your topic branch and drop the commits you didn't want to pull in.

5.3.6 Tagging Your Releases

When you've decided to create a release, you'll probably want to create a tag as I discussed in Chapter 2 so you can re-create that release at any point going forward. If you decide to sign the tag as the maintainer, the tagging may look something like

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

If you do sign your tags, you'll have the problem of distributing the public PGP key you used to sign your tags. The maintainer of the Git project has solved this issue by including his public key as a blob in the repository and then adding a tag that points directly to that content. To do this, figure out which key you want by running `gpg --list-keys`.

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid                  Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Then, directly import the key into the Git database by exporting it from `gpg` and piping the key through `git hash-object`, which writes a new blob with the key's contents into Git and gives you back the SHA-1 hash of the blob.

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Now that you have the contents of your key in Git, create a tag that points directly to it by specifying the new SHA-1 hash that the `hash-object` command gave you.

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

If you run `git push --tags`, the `maintainer-pgp-pub` tag will be shared with everyone. If anyone wants to verify a tag, they can directly import your PGP key by pulling the blob directly out of the database and importing it into GPG.

```
$ git show maintainer-pgp-pub | gpg --import
```

They can use that key to verify all your signed tags. Also, if you include instructions in the tag message, running `git show <tag>` will give the end user more specific instructions about tag verification.

5.3.7 Generating a Build Number

Because Git doesn't give each commit a monotonically increasing number, like 'v123' or the equivalent, if you want a human-readable name to go with a commit, run `git describe` on that commit. This shows the name of the nearest tag with the number of commits after that tag and a partial SHA-1 hash of the commit you're describing.

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

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

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

5.3.8 Preparing a Release

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

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

That tarball contains the latest snapshot of your project under a project directory. You also create a zip archive in much the same way, but by passing the `--format=zip` option to `git archive`.

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

You now have a nice tarball and a zip archive of your project release that you upload to your website or e-mail to people.

5.3.9 The Shortlog

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

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.1
      Regenerated gemspec for version 1.0.2
```

You get a clean summary of all the commits since v1.0.1, grouped by author, that you can e-mail to your list.

5.4 Summary

You should feel fairly comfortable contributing to a project managed by Git as well as maintaining your own project or integrating other users' contributions. Congratulations on being an effective Git developer! In the next chapter, you'll learn more powerful tools and tips for dealing with complex situations, which will truly make you a Git master.

Chapter 6

Git Tools

By now, you've learned most of the day-to-day commands and workflows that you need to manage and maintain a Git source code repository. You've learned the basic tasks of tracking and committing files, and you've harnessed the power of the staging area, lightweight topic branching, and merging.

Now you'll explore a number of very powerful things that Git can do that you may not necessarily use on a day-to-day basis but that you may need one day.

6.1 Commit Selection

Git allows you to specify specific commits or a range of commits in several ways.

6.1.1 Single Commit

You can obviously refer to a commit by its SHA-1 hash, but there are also more human-friendly ways. This section outlines the various ways you can refer to a single commit.

6.1.2 Short SHA-1 Hash

Git is smart enough to figure out what commit you're referring to if you only provide the first few characters of its SHA-1 hash, as long as the partial hash is at least four characters long and unambiguous — that is, only one object in the current repository begins with that partial SHA-1 hash.

For example, to see a specific commit, suppose you run `git log` to identify the commit where you made a certain change.

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated 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

    added some blame and merge stuff
```

In this case, let's say `1c002dd....` is the commit you're interested in. For `git show` to show that commit, the following commands are equivalent (assuming the shorter versions are unambiguous):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git can recognize a short, unique representation for your SHA-1 hashes. If you pass `--abbrev-commit` to `git log`, the output will contain short unique hash values. This option defaults to showing seven character hash values but makes the hashes longer if necessary to keep them unambiguous.

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

Generally, eight to ten characters are more than enough for a SHA-1 hash to be unique within a project. One of the largest Git projects, the Linux kernel, is beginning to need 12 characters out of the possible 40 for SHA-1 hashes to stay unique.

6.1.3 A SHORT NOTE ABOUT SHA-1 HASHES

You might become concerned at some point that you will, by random happenstance, attempt to place two different objects into your repository that hash to the same SHA-1 hash. What then?

If you do happen to commit an object that hashes to the same SHA-1 hash as an object already in your repository, Git will see that the new object is already in your Git repository

and assume there's no need to write it again. If you try to check out that object again, you'll always get the contents of the first object.

However, you should be aware of how ridiculously unlikely this scenario is. A SHA-1 hash is 20 bytes or 160 bits. The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about 2^{80} (the formula for determining collision probability is `p = (n(n-1)/2) * (1/2^160))`. 2^{80} is 1.2×10^{24} or 1 million billion billion. That's 1,200 times the number of grains of sand on earth.

Here's an example to give you an idea of what it would take to get a SHA-1 hash collision. If all 6.5 billion humans on Earth were programming, and every second each one was producing code that was the equivalent of the entire Linux kernel history (1 million Git objects) and pushing it into one enormous Git repository, it would take 5 years until that repository contained enough objects to have a 50% probability of a single SHA-1 hash object collision. A higher probability exists that every member of your programming team will be attacked and killed by wolves in unrelated incidents on the same night.

6.1.4 Branch References

The most straightforward way to specify a commit requires that the commit have a branch reference pointed at it. Then, you can use a branch name in any Git command that expects a commit object or SHA-1 hash. For instance, to show the last commit object on a branch, the following commands are equivalent, assuming that the `topic1` branch points to `ca82a6d`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

To see which specific SHA-1 hash a branch points to, use a Git plumbing tool called `git rev-parse`. See Chapter 9 for more information about plumbing tools. Basically, `git rev-parse` exists for lower-level operations and isn't designed to be used in day-to-day operations. However, it can be helpful sometimes when you need to see what's really going on. Here, run `git rev-parse` on your branch.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

6.1.5 RefLog Shortnames

Git keeps a reflog — a log of where your HEAD and branch references have been. You can see your reflog by running `git reflog`.

```
$ git reflog
```

```
734713b... HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970... HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd... HEAD@{2}: commit: added some blame and merge stuff
1c36188... HEAD@{3}: rebase -i (squash): updating HEAD
95df984... HEAD@{4}: commit: # This is a combination of two commits.
1c36188... HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5... HEAD@{6}: rebase -i (pick): updating HEAD
```

Every time your branch tip is updated for any reason, Git stores the new value in this temporary log. You can also specify older commits as well. If you want to see the fifth prior value of the HEAD of your repository, use the `@{n}` syntax that you see in the reflog output.

```
$ git show HEAD@{5}
```

You can also use this syntax to see where a branch was some specific amount of time ago. For instance, to see where your `master` branch was yesterday, run

```
$ git show master@{yesterday}
```

That shows you where the branch tip was yesterday. This technique only works for data that's still in your reflog, so you can't use it to look for commits older than a few months.

To see reflog information formatted like `git log` output, run `git log -g`.

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

It's important to note that the reflog information is strictly local — it's a log of what you've done in your repository. The references won't be the same in someone else's copy of the repository. Right after you initially clone a repository, you'll have an empty reflog, since no activity has occurred yet in your repository. Running `git show HEAD@{2.months.ago}` will work only if you cloned the project at least two months ago. If you cloned it five minutes ago, you'll see no results.

6.1.6 Ancestry References

The other main way to specify a commit is via its ancestry. If you place a `^` at the end of a reference, Git resolves it to mean the parent of that commit. Suppose you look at the history of your project.

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

Then you can see the previous commit by specifying `HEAD^`, which means "the parent of HEAD".

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

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

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
```

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

    added some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

The other main ancestry specification is the `~`. This also refers to the first parent, so `HEAD~` and `HEAD^` are equivalent. The difference becomes apparent when you specify a number. `HEAD~2` means "the first parent of the first parent," or "the grandparent" — it traverses the first parents the number of times you specify. For example, in the history listed earlier, `HEAD~3` would be

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

This can also be written `HEAD^^^`, which again is the first parent of the first parent of the first parent.

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

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

6.1.7 Commit Ranges

Now that you can specify individual commits, let's see how to specify ranges of commits. This is particularly useful for managing your branches — if you have a lot of branches, use

range specifications to answer questions such as, “What work is on this branch that I haven’t yet merged into my main branch?”

Double Dot

The most common range specification is the double-dot syntax. This specifies a range of commits that are reachable from one commit but aren’t reachable from another. For example, say you have a commit history that looks like Figure 6-1.

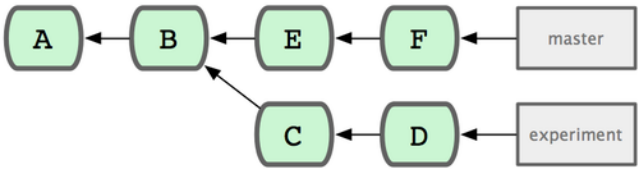

Figure 6.1: Example history for range selection.

You want to see what’s in your `experiment` branch that hasn’t yet been merged into your `master` branch. Ask Git to show a list of just those commits with `master..experiment`. That means “all commits reachable by `experiment` that aren’t reachable by `master`.” For the sake of brevity and clarity in these examples, I’ll use the letters of the commit objects from the diagram in place of the actual log output.

```
$ git log master..experiment
D
C
```

If, on the other hand, you want to see the opposite — all commits in `master` that aren’t in `experiment` — reverse the branch names. `experiment..master` shows you everything in `master` not reachable from `experiment`.

```
$ git log experiment..master
F
E
```

This is useful for keeping the `experiment` branch up to date and for previewing what you’re about to merge in. Another very common use of this syntax is to see what you’re about to push to a remote.

```
$ git log origin/master..HEAD
```

This command shows any commits in your current branch that aren’t in the `master` branch on your `origin` remote. If you run `git push` and your current branch is tracking `origin/master`, the commits listed by `git log origin/master..HEAD` are those that will

be transferred to the server. If you leave off one side of the specification, Git assumes the missing side is HEAD. For example, you get the same results as in the previous example by running `git log origin/master...`

Multiple Points

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

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

This is nice because with this syntax you can specify more than two references in your query, which you can't do with the double-dot syntax. For instance, to see all commits reachable from `refA` or `refB`, but not from `refC`, run one of

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

This makes for a very powerful revision query system that should help you figure out what's in your branches.

Triple Dot

The last major range-selection form is the triple-dot syntax, which specifies all the commits that are reachable by either of two references, but not by both. Look back at the example commit history in Figure 6-1. To see what's in `master` or `experiment` but not any common references, run

```
$ git log master...experiment
F
E
D
C
```

Again, this produces normal `git log` output but shows you only the commit information for those four commits, appearing in the traditional commit date ordering.

A common option to use with `git log` in this case is `--left-right`, which shows which side of the range each commit is in. This helps make the data more useful.

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

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

6.2 Interactive Staging

Git comes with a couple of scripts that make some command-line tasks easier. Here, you'll see a few interactive commands that can help you easily craft commits to include only certain combinations and parts of files. These tools are very helpful if you modify a bunch of files and then decide that you want those changes to be in several focused commits rather than one big messy commit. This way, you can make sure your commits are logically separate and can be easily reviewed by the developers working with you. If you run `git add` with the `-i` or `--interactive` option, Git goes into an interactive mode, displaying something like

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now>
```

You can see that this command shows a much different view of your staging area — basically the same information you get from `git status` but a bit more succinct and informative. It lists the changes you've staged on the left and the unstaged changes on the right.

After this comes a Commands section. Here you can do a number of things, including staging files, unstaging files, staging parts of files, adding untracked files, and seeing diffs of what has been staged.

6.2.1 Staging and Unstaging Files

If you type `2` or `u` at the `What now>` prompt, you're prompted for which files to stage.

```
What now> 2
           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, type their 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 name means the file is selected to be staged. If you just type Enter at the `Update>>` prompt, Git takes what you selected and stages it for you.

```
Update>>
updated 2 paths

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Now you see that TODO and index.html are staged and simplegit.rb is still unstaged. To unstage TODO at this point, enter the `3` or `r` (for revert) option.

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 3
           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 see that you've unstaged TODO.

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           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, use the `6` or `d` (for diff) command. It shows a list of staged files, and you can select the files for which you would like to see the diffs between what you've staged and what's in the most recent commit. This is much like running `git diff --cached` on the command line.

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 6
           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 manage your staging area a little more easily.

6.2.2 Staging Patches

It's also possible for Git to stage certain parts of files and not others. For example, if you make two changes to simplegit.rb and want to stage one of them and not the other, doing so is very easy in Git. From the interactive prompt, type `5` or `p` (for patch). Git will ask which files to partially stage. Then, for each section of the selected files, it displays hunks of the file diff and asks 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` to stage each hunk, but staging all of them in certain files or postponing a hunk decision until later can be helpful too. If you stage one part of a file and leave another part unstaged, your status output looks like

```
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 simplegit.rb is interesting. It shows 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 script and run `git commit` to commit the partially staged files.

You don't need to be in interactive add mode to do partial-file staging. You can start the same script by running `git add -p` or `git add --patch` on the command line.

6.3 Stashing

Often, when you've been working on part of a project, your working directory gets into a messy state, and you want to switch branches for a bit to work on something else. The problem is, you don't want to commit half-done work just so you can return to this state later. The answer to this dilemma is the `git stash` command.

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes to which you can revert at any time.

6.3.1 Stashing Your Work

To demonstrate, start working on a couple of files in a project and stage one of them. Run `git status` to see your state.

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
#      modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
```

Now, you want to switch to a different branch, but you don't want to commit what you've been working on yet. Instead, stash the changes. To push a new stash onto your stack, run `git stash`.

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Your working directory is now clean.

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

At this point, you can easily start working on a different branch. Your changes are stored on your stack. To see the stashes you've stored, run `git stash list`.

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

In this case, you already did two stashes previously, so are now three different stashed states. Revert to the one you just stashed by using the command shown in the help output of the original stash command, `git stash apply`. To revert to one of the older stashes, name it like this: `git stash apply stash@{2}`. If you don't specify a stash, Git tries to revert to the most recent stash.

```
$ git stash apply
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   index.html
#      modified:   lib/simplegit.rb
#
```

You can see that Git restored the uncommitted files from when you saved the stash. In this case, you had a clean working directory when you tried to revert from the stash, and you tried to revert on the same branch you saved from. But having a clean working directory and reverting it on the same branch aren't necessary to successfully revert a stash. You can save a stash on one branch, switch to another branch later, and try to reapply the changes. You can also have modified and uncommitted files in your working directory when you revert a stash — Git shows merge conflicts if anything no longer applies cleanly.

The changes to your files were reapplied, but any files you staged before weren't restaged. To do that, you must run `git stash apply` with the `--index` option to try to reapply the staged changes.

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
```

`git stash apply` only tries to revert the stashed work — the stash remains on your stack. To remove a stash, run `git stash drop` with the name of the stash to remove.

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
$ git stash drop stash@{0}
```

```
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

You can also run `git stash pop` to revert the stash and then immediately remove it from your stack.

6.3.2 Un-applying a Stash

In some cases you might want to revert stashed changes, do some work, but then stash those changes that originally came from the stash. Git does not include a `stash unapply` command, but it's possible to achieve the same effect by simply retrieving the patch associated with a stash and applying it in reverse.

```
$ git stash show -p stash@{0} | git apply -R
```

Again, if you don't specify a stash, Git assumes the most recent stash.

```
$ git stash show -p | git apply -R
```

You may want to create an alias to effectively add a `stash-unapply` option to `git`. For example

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash
$ #... work work work
$ git stash-unapply
```

6.3.3 Creating a Branch from a Stash

If you stash some work, leave it for a while, and continue on the branch from which you stashed the work, you may have a problem reverting the work. If reverting tries to modify a file that you've since modified, you'll get a merge conflict that you'll have to try to resolve. If you want an easier way to test the stashed changes again, run `git stash branch`, which creates a new branch, checks out the commit you were on when you stashed your work, reverts your work there, and then drops the stash if it applies successfully.

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

This is a nice shortcut to easily revert stashed work to work on in a new branch.

6.4 Rewriting History

Many times, when working with Git, you want to revise your commit history. One of the great things about Git is that it allows you to make decisions at the last possible moment. You can decide what files go into which commits right before you commit by only putting certain files in the staging area, you can decide that you don't mean to be working on something yet with the stash command, and you can rewrite previous commits so they look like they happened differently. This can involve changing the order of the commits, changing messages or modifying the files in a commit, squashing together or splitting apart commits, or removing commits entirely — all before you share your work with others.

In this section, I'll cover how to accomplish these very useful tasks so that you can make your commit history look the way you want before you share your repository.

6.4.1 Changing the Last Commit

Changing your last commit is probably the most common rewriting of history that you'll do. You'll often want to change two basic things in your last commit — the commit message, or the snapshot you just recorded by adding, changing, and removing files.

If you only want to modify your last commit message, it's very simple.

```
$ git commit --amend
```

That drops you into your text editor, which will have your last commit message in it, ready to be modified. When you save your message and exit the editor, it writes a new commit containing the new message and makes the commit your new last commit.

If you've committed and then want to change the snapshot you committed by adding or changing files, possibly because you forgot to add a newly created file when you originally committed, the process works basically the same way. Stage the changes you want by running `git add` on a modified file or `git rm` on a tracked file, and the subsequent `git`

`commit --amend` takes your current staging area and makes it the snapshot for the new commit.

Be careful with this technique because `git commit --amend` changes the SHA-1 hash of the commit. It's like a very small rebase — don't amend your last commit if you've already pushed.

6.4.2 Changing Multiple Commit Messages

To modify a commit that's farther back in your history, you must use more complex tools. Git doesn't have a modify-history tool, but you can rebase to move a series of commits onto the branch they were originally based on instead of moving them to another branch. 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 run rebase interactively by running `git rebase -i`. You must indicate how far back you want to rewrite commits by telling the command which commit to rebase onto.

For example, to change any of the last three commit messages, supply the parent of the last commit to edit, which is `HEAD~2^` or `HEAD~3`, as an argument to `git rebase -i`. 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 referring to a commit four commits back — 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 included in the range `HEAD~3..HEAD` will be rewritten whether or not you change the message. 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.

Running this command opens your text editor with a list of commits in the buffer that look something like

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

It's important to note that these commits are listed in the opposite order from what you normally see using the `git log` command. If you run `git log`, you see something like

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Again, notice the reverse order. The interactive rebase creates a script that it's going to run. It starts at the commit you specify on the command line (`HEAD~3`) and replays the changes introduced in each of these commits, from top to bottom. It lists the oldest at the top, rather than the newest, because that's the first one it will replay.

You need to edit the script so that it stops at the commit you want to edit. To make this happen, change the word `pick` to the word `edit` in each of the commits you want the script to stop after. For example, to modify only the third commit message, change the file to look like

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

When you save the file and exit the editor, Git rewinds back to the last commit in that list and shows the following message:

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

These instructions tell you exactly what to do. Type

```
$ git commit --amend
```

Change the commit message, and exit the editor. Then, run

```
$ git rebase --continue
```

This command applies the other two commits automatically, and then you're done. If you change `pick` to `edit` on more lines, you can repeat these steps for each commit you want to change. Each time Git will stop, let you amend the commit, and then continue when you're finished.

6.4.3 Reordering Commits

You can also use interactive rebases to entirely reorder or remove commits. To remove the "added cat-file" commit and change the order in which the other two commits appear, change the rebase script from this

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

to this.

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

When you save the file and exit the editor, Git rewinds your branch to the parent of these commits, applies `310154e` and then `f7f3f6d`, and then stops. You effectively change the order of those commits and remove the "added cat-file" commit completely.

6.4.4 Squashing Commits

It's also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool. The script puts helpful instructions in the rebase message.

```
#
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
```

```
# However, if you remove everything, the rebase will be aborted.
#
```

If, instead of `pick` or `edit`, you specify `squash`, Git applies both that change and the change directly before it, and makes you merge the commit messages together. So, to make a single commit from these three commits, make the script look like

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

When you save and exit the editor, Git applies all three changes and then puts you back into the editor to merge the three commit messages.

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

When you save that file, you have a single commit that includes the changes from all three previous commits.

6.4.5 Splitting a Commit

Splitting a commit undoes a commit and then partially stages and commits as many times as the number of commits you want to end up with. For example, suppose you want to split the middle commit of your three commits. Instead of “updated README formatting and added blame”, you want to split it into two commits — “updated README formatting” for the first commit, and “added blame” for the second. You do that in the `git rebase -i` script by changing the command on the commit you want to split to `edit`.

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

When you save the file and exit the editor, Git rewinds to the parent of the first commit in your list, applies the first commit (`f7f3f6d`), applies the second (`310154e`), and exits. Then, do a mixed reset of that commit by running `git reset HEAD^`, which effectively undoes that commit and leaves the modified files unstaged. Now you can take the changes that have been reset, and create multiple commits out of them. Simply stage and commit files until you have several commits, and run `git rebase --continue` when you're done.

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git applies the last commit (`a5f4a0d`) in the script, making your history look like

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Once again, this changes the SHA-1 hashes of all the commits in your list, so make sure no commit shows up in that list that you've already pushed to a shared repository.

6.4.6 The Nuclear Option: filter-branch

There's another history-rewriting option to rewrite a larger number of commits in some scriptable way — for instance, changing your e-mail address globally or removing a file from every commit. The command is `git filter-branch`. It can rewrite huge swaths of history, so you probably shouldn't use it unless your project isn't yet public and other people haven't based work off the commits you're about to rewrite. However, it can be very useful. You'll learn a few of the common uses now so you can get an idea of some of the things it's capable of.

Removing a File from Every Commit

This occurs fairly commonly. Someone accidentally commits a huge binary file after a thoughtless `git add .`, and you want to remove it in every commit. Perhaps you accidentally committed a file that contained a password, and you want to make your project open source. `git filter-branch` is the tool you probably want to use to scrub your entire history. To remove a file named `passwords.txt` from your entire commit history, run `git filter-branch --tree-filter`.

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

The `--tree-filter` option runs the specified command after each checkout of the project and then recommits the results. In this case, you remove `passwords.txt` from every snapshot, whether or not it exists. To remove all accidentally committed editor backup files, run something like

```
$ git filter-branch --tree-filter "rm -f *~" HEAD
```

You'll be able to watch Git rewriting trees and commits, and then move the branch pointer at the end. It's generally a good idea to do this in a testing branch and then hard-reset your master branch after you've determined the outcome is what you really want. To run `git filter-branch` on all your branches, add `--all` to the command.

Making a Subdirectory the New Root

Suppose you've done an import from another source control system and have subdirectories that make no sense (`trunk`, `tags`, and so on). To make the `trunk` subdirectory be the new project root for every commit, `git filter-branch` can help you do that, too.

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Now your new project root is what was in the `trunk` subdirectory. Git will also automatically remove commits that didn't affect the subdirectory.

Changing E-Mail Addresses Globally

Another common case is that you forgot to run `git config` to set your name and e-mail address before you started working, or perhaps you want to open-source a project at work and change all your work e-mail addresses to your personal address. In any case, you can change e-mail addresses in multiple commits in a batch with `git filter-branch` as well. Be careful to change only the e-mail addresses that are yours, so use `--commit-filter`.

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
```

```
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

This goes through and rewrites every commit to have your new address. Because commits contain the SHA-1 hashes of their parents, this command changes every commit SHA-1 hash in your history, not just those having the matching e-mail address.

6.5 Debugging with Git

Git also provides a couple of tools to help you debug issues in your projects. Because Git is designed to work with nearly any type of project, these tools are pretty generic, but they can often help hunt for a bug when things go wrong.

6.5.1 File Annotation

If you track down a bug in your code and want to know when it was introduced and why, file annotation is often your best tool. It shows what commit was the last to modify each line in any file. So, if you see that a method in your code is buggy, annotate the file by running `git blame` to see when each line of the method was last edited, and by whom. This example uses the `-L` option to limit the output to lines 12 through 22.

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 12)  def show(tree = 'master')
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 13)  command("git show #{tree}")
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 14)  end
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 16)  def log(tree = 'master')
79eaf55d (Scott Chacon  2008-04-06 10:15:08 -0700 17)  command("git log #{tree}")
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 18)  end
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20)  def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)  command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22)  end
```

Notice that the first field in the output is the partial SHA-1 hash of the commit that last modified that line. The next two fields are values extracted from that commit—the author name and the date of that commit — so you can easily see who modified that line and when. After that come the line number and the contents of the file. Also note the `^4832fe2` commit lines, which show that those lines were in this file's original commit. That commit is when

this file was first added to this project, and those lines have been unchanged ever since. This is a tad confusing, because now you've seen at least three different ways that Git uses `^` to modify a commit SHA-1 hash.

Another cool thing about Git is that it doesn't track file renames explicitly. It records the snapshots and then tries to figure out what was renamed implicitly, after the fact. One of the interesting results of this is that you can ask Git to figure out all sorts of code movement as well. If you pass `-C` to `git blame`, Git analyzes the file you're annotating and tries to figure out where snippets of code within it originally came from if they were copied from elsewhere. Recently, I was refactoring a file named `GITServerHandler.m` into multiple files, one of which was `GITPackUpload.m`. By running `git blame` with the `-C` option, I could see where sections of the code originally came from.

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)        //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)          NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)        GITCommit *commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)        //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)          if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

This is really useful. Normally, the commit you see as the original commit is from when you copied the code over, because that was the first time you touched those lines in this file. Git tells you the original commit where you wrote those lines, even if they're from another file.

6.5.2 Binary Search

Annotating a file helps if you know where the issue is to begin with. If you don't know what caused the problem, and there have been dozens or hundreds of commits since when the code last worked, turn to `git bisect` for help. This command does a binary search through your commit history to help identify as quickly as possible which commit introduced a problem.

Let's say you just pushed out a release of your code to a production environment, and you're getting bug reports about something that wasn't happening in your development environment. You have no idea what's going wrong. You go back to your code, and it turns out you can reproduce the issue, but you still can't figure out what's wrong. You can bisect the code to find out. First, run `git bisect start` to get things going, and then use `git bisect`

`bad` to tell the system that the current commit you're on is broken. Then, tell Git when the last known good state was by running `git bisect good [good_commit]`.

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git figured out that about 12 commits came between the commit you marked as the last good commit (v1.0) and the current bad version, and it checked out the middle commit for you. At this point, run your test to see if the issue exists in this commit. If it does, then it was introduced sometime before. If it doesn't, then the problem was introduced sometime after. If it turns out there's no issue here, tell Git by running `git bisect good` again and continue your journey.

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Now you're on another commit, halfway between the one you just tested and your bad commit. When you run your test again you find that this commit is broken, so tell Git by running `git bisect bad`.

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

This commit is fine, and now Git has all the information it needs to determine where the issue was introduced. It tells you the SHA-1 hash of the first bad commit and shows some of the commit information and which files were modified in that commit so you can figure out what may have introduced this bug.

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing
```

```
:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

When you're finished, run `git bisect reset` to reset your HEAD to where you were before you started, or you'll end up in a weird state.

```
$ git bisect reset
```

This is a powerful tool that can help you check hundreds of commits for a mysterious bug in minutes. In fact, if you have a script that returns 0 if the project code is good or non-0 if the project code is bad, you can fully automate `git bisect`. First, again tell Git the scope of the bisect by providing the known bad and good commits. Do this by including them on the `bisect start` command line, placing the known bad commit first and the known good commit second.

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

This automatically runs `test-error.sh` on each checked-out commit until Git finds the first broken commit. You can also run something like `make`, `make tests`, or whatever you have that runs automated tests for you.

6.6 Submodules

Sometimes you work on a project that itself depends on another project. Perhaps the other project is a library that someone else developed or that you're developing separately and using in multiple parent projects. A common issue arises in these scenarios — how 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 web site 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 install the library using CPAN or a Ruby gem, or copy the library source code into your own project. The issue with including the source code is that it's difficult to do any customization of the library, and using the library is a nuisance because you need to make sure every client has that library installed. The issue with incorporating the library code into your own project is that any custom changes you make are difficult to merge when changes to the library become available from its developer.

Git addresses this issue with submodules. Submodules allow you to keep one Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

6.6.1 Starting with Submodules

Suppose you want to add the Rack library (a Ruby web server gateway interface) to your project, possibly making your own changes to it, but you want to continue to merge changes from its developers. The first thing to do is to clone the external repository into your subdirectory. Add external projects as submodules with the `git submodule add` command.

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
```

Now you have the Rack project in a subdirectory named `rack` within your project. You can `cd` to that subdirectory, make changes, add your own writable remote repository to push your changes to, fetch and merge from the original repository, and more. If you run `git status` right after you add the submodule, you see two things.

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      new file:   .gitmodules
#      new file:   rack
#
```

First, you notice the `.gitmodules` file. This is a configuration file that stores the mapping between the project's URL and the local subdirectory where you put it.

```
$ cat .gitmodules
[submodule "rack"]
      path = rack
      url = git://github.com/chneukirchen/rack.git
```

If you have multiple submodules, this file will have multiple entries. It's important to note that Git tracks this file along 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.

The other listing in the `git status` output is the `rack` entry. If you run `git diff` on that, you see something interesting.

```
$ git diff --cached rack
diff --git a/rack b/rack
new file mode 160000
index 0000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

Although `rack` 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 records it as a particular commit from that repository. When you make changes and commit in that subdirectory, the superproject notices that the HEAD there has changed and records the exact commit you're currently working off of. That way, when others clone this project, they can re-create the environment exactly.

This is an important point with submodules. Reference them as the exact commit they're at. You can't reference a submodule at `master` or some other symbolic reference.

When you commit, you see something like

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
```

Notice the 160000 mode for the `rack` entry. That's a special mode in Git that means you're recording a commit as a directory entry rather than as a subdirectory or a file.

You can treat the `rack` directory as a separate project and then update your superproject from time to time with a pointer to the latest commit in that subproject. All the Git commands work independently in the two directories.

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:03:56 2009 -0700

    first commit with submodule rack
$ cd rack/
```

```
$ git log -1
commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date:   Wed Mar 25 14:49:04 2009 +0100

    Document version change
```

6.6.2 Cloning a Project with Submodules

Here you clone a project containing a submodule. When you receive such a project, you get the directories that contain submodules, but none of the files yet.

```
$ git clone git://github.com/schacon/myproject.git
Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
total 8
-rw-r-r--  1 schacon  admin   3 Apr  9 09:11 README
drwxr-xr-x  2 schacon  admin  68 Apr  9 09:11 rack
$ ls rack/
$
```

The `rack` 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 files from that project and check out the appropriate commit listed in your superproject.

```
$ git submodule init
Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afcf433'
```

Now your `rack` subdirectory is in the exact state it was in when you committed earlier. If another developer makes changes to the rack code and commits, and you pull that reference and merge it, you get something a bit odd.

```
$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
# On branch 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:   rack
#
```

You merged in what is basically a change to the pointer for your submodule. But this doesn't update the code in the submodule directory, so it looks like your working directory is in a dirty state.

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
+++ b/rack
@@ -1 +1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433


$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
   08d709f..6c5e70b  master     -> origin/master
Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

You have to do this every time you pull down a submodule change in the main project. It's strange, but it works.

One common problem happens when a developer makes a change locally in a submodule but doesn't push it to a public server. Then, they commit a pointer to that non-public state and push the superproject. When other developers try to run `git submodule update`, the submodule system can't find the referenced commit because it exists only on the first developer's system. If that happens, you see an error like

```
$ git submodule update
fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'rack'
```

You have to see who last changed the submodule.

```
$ git log -1 rack
commit 85a3eee996800fcfa91e2119372dd4172bf76678
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:19:14 2009 -0700

    added a submodule reference I will never make public. hahahahaha!
```

Then, you e-mail that guy and yell at him.

6.6.3 Superprojects

Sometimes, developers want to get a combination of a large project's subdirectories, depending on what team they're on. This is common if you're coming from CVS or Subversion, where you've defined a module or collection of subdirectories, and you want to keep this type of workflow.

A good way to do this in Git is to make each of the subdirectories a separate Git repository and then create superproject Git repositories that contain multiple submodules. A benefit of this approach is that you can more specifically define the relationships between the projects with tags and branches in the superprojects.

6.6.4 Issues with Submodules

Using submodules isn't without hiccups, however. First, be careful when working in the submodule directory. When you run `git submodule update`, it checks out the specific version of the project, but not within a branch. This is called having a detached HEAD — it means HEAD points directly to a commit, not to a symbolic reference. The problem is that you generally don't want to work in a detached HEAD environment because it's easy to lose changes. If you run an initial `git submodule update`, commit in that submodule directory without creating a branch to work in, and then run `git submodule update` again from the superproject without committing in the meantime, Git will overwrite your changes without

telling you. Technically you won't lose the work, but you won't have a branch pointing to it, so it will be difficult to retrieve.

To avoid this issue, create a branch when you work in a submodule directory with `git checkout -b work` or something equivalent. When you update the submodule a second time, it will still revert your work, but at least you have a pointer to get back to.

Switching branches containing submodules can also be tricky. If you create a new branch, add a submodule, and then switch back to a branch without that submodule, you still have the submodule directory as an untracked directory.

```
$ git checkout -b rack
Switched to a new branch "rack"
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#      rack/
```

You have to either move the submodule directory out of the way or remove it, in which case you have to clone it again when you switch back. You may lose local changes or branches that you didn't push.

The last 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, be careful or Git will get angry at you. Assume that you have the `rack` 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 rack/
$ git submodule add git@github.com:schacon/rack.git rack
'rack' already exists in the index
```

You have to unstage the `rack` directory first. Then, add the submodule.

```
$ git rm -r rack
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/testsub/rack/.git/
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 | 88 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
```

Suppose you did that in a branch. If you try to switch back to a branch in which those files are still in the actual tree rather than a submodule you get

```
$ git checkout master
error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

You have to move the `rack` submodule directory out of the way before you can switch to a branch that doesn't contain it.

```
$ mv rack /tmp/
$ git checkout master
Switched to branch "master"
$ ls
README rack
```

Then, when you switch back, you get an empty `rack` directory. You can either run `git submodule update` to reclone, or move your `/tmp/rack` directory back into the empty directory.

6.7 Subtree Merging

Now that you've seen the difficulties of dealing with submodules, let's look at an alternate way to solve the same problem. When Git does a merge, it looks at what it has to merge and then chooses an appropriate merging strategy. 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't handle complex three-way merges — for example, more than one common ancestor — it can only handle merging two branches. The octopus merge can handle multiple branches but is more cautious about avoiding difficult conflicts, so it's chosen as the default strategy for merging more than two branches.

However, there are other strategies to choose from as well. One of them is the *subtree* merge, which you can use to deal with the subproject issue. Here you'll see how to do the same `rack` subdirectory 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.

First add the Rack application to your project. 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 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 by running `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 is updated, pull in upstream changes by switching to that branch and pulling.

```
$ git checkout rack_branch
$ git pull
```

Then, merge those changes back into your master branch. Running `git merge -s subtree` works fine but Git will also merge the histories together, which you probably don't want. To pull in the changes and pre-populate 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's 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 `git diff` command. Instead, run `git diff-tree` with the branch you want to compare to.

```
$ git diff-tree -p rack_branch
```

Or, to compare what's in your `rack` subdirectory with what was in the `master` branch on the server the last time you fetched, run

```
$ git diff-tree -p rack_remote/master
```

6.8 Summary

You've seen a number of advanced tools that allow more precise manipulation of your commits and staging area. When you notice issues, you should be able to easily figure out what commit introduced them, when they were committed, and who committed them. You've also learned a few ways to use subprojects in your project. At this point, you should be able to comfortably do most of the things in Git that you'll need every day.

Chapter 7

Customizing Git

So far, I've covered the basics of how Git works and how to use it, and I've introduced a number of Git tools that help you use it efficiently. In this chapter, I'll go through some steps that you can perform to make Git operate in a more personalized fashion. I'll introduce several important configuration settings and the hooks system. With these, it's easy to get Git to work exactly the way you, your company, or your group needs it to.

7.1 Git Configuration

As you briefly saw in the Chapter 1, you modify Git configuration settings with the `git config` command. One of the first things you did was to set 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 settings.

You saw some simple Git configuration details in Chapter 1, but I'll go over them again quickly here. Git uses a collection of configuration files for storing settings. The first place Git looks for these settings is in the `/etc/gitconfig` file, which contains settings that apply to every user on the system and in all repositories. If you run `git config --system`, it reads and writes from this file.

The next place Git looks is the `~/.gitconfig` file, which is user specific. Git reads and writes to this file when you run `git config --global`.

Finally, Git looks for configuration settings in the `.git/config` file in the repository you're currently using. These settings are specific to that single repository.

Each level overwrites settings in the previous level, so settings in `.git/config` trump those in `/etc/gitconfig`, for instance. You can also set these settings manually by editing the file and inserting the correct setting, but it's generally easier to run `git config`.

7.1.1 Basic Client Configuration

The configuration settings recognized by Git fall into two categories: client side and server side. The majority of the settings are client side — configuring only your preferences.

Although tons of settings are available, I'll only cover the few that either are commonly used or can significantly affect your workflow. Many settings are useful only in edge cases that I won't go over. To see a list of all the settings the version of Git you're running recognizes, run

```
$ git config --help
```

The manual page for `git config` lists all the available settings in quite a bit of detail.

core.editor

By default, Git uses whatever you've set in your shell EDITOR variable or else falls back to the Vi editor to create and edit commit and tag messages. To modify that default, use the `core.editor` setting.

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

Now, no matter what's set in your shell EDITOR variable, Git will fire up Emacs to edit messages.

commit.template

If you set commit.template to the path of a file on your system, Git will use the contents of that file as the default message when you commit. For instance, suppose you create the template file `$HOME/.gitmessage.txt` that looks like

```
subject line

what happened

[ticket: X]
```

To tell Git to use the contents of this file as the default message that appears in your editor when you run `git commit`, set the `commit.template` configuration setting.

```
$ git config --global commit.template $HOME/.gitmessage.txt
$ git commit
```

Then, your editor buffer will look something like this when you run `git 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 creating a template that satisfies that policy and configuring Git to use it by default can help increase the chances of that policy being followed.

core.pager

The core.pager setting determines what pager Git uses to page output from commands such as `git log` and `git diff`. You can set it to `more` or to your favorite pager (the default is `less`), or you can turn it off by setting it to a blank string.

```
$ git config --global core.pager ''
```

If you do that, Git won't page the output of any command, no matter how long the output 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 GPG signing key like so.

```
$ git config --global user.signingkey <gpg-key-id>
```

Now you can sign tags without having to specify your key every time when you run `git tag`.

```
$ git tag -s <tag-name>
```

core.excludesfile

Put patterns in your project's `.gitignore` file so that Git ignores untracked files that match the patterns, as discussed in Chapter 2. However, if you want to place these patterns in a file outside of your project, tell Git the location of that file with the `core.excludesfile` setting. Simply set it to the path of a file with content similar to what's in `.gitignore`.

help.autocorrect

If you mistype a command in Git, you see something like

```
$ 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 suggested command, but only if there's just one matched possibility.

7.1.2 Colors in Git

Git can color what it sends to your screen, which can make it easier to understand the output. A number of settings can help you set the colors to your liking.

color.ui

Git automatically colors most of its output. You can get very specific about what you want colored and how. To turn on default coloring, set `color.ui` to true.

```
$ git config --global color.ui true
```

With that setting, Git colors its output if the output goes to a screen. Other possible settings are false, which never colors the output, and always, which colors all the time, even if you're redirecting Git commands to a file or piping them to another command.

You'll rarely set `color.ui = always`. In most scenarios, if you want color in your redirected output, instead pass the `--color` flag to the Git command to force it to use colors. The `color.ui = true` setting is almost always what you'll want.

```
color.*
```

To be more specific about which commands produce colors and how, Git contains verb-specific color 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 to set specific colors for parts of the output. For example, to set the meta information in diff output to blue foreground, black background, and bold text, 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.

7.1.3 External Merge and Diff Tools

Although you've been using Git's internal implementation of diff, you can use an external tool instead. You can also use 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, it's free, and it works on all major platforms. I'll use Mac and Linux path names in the examples.

Download P4Merge here.

http://www.perforce.com/perforce/downloads/component.html

To begin, create external wrapper scripts to run your commands. I'll use the Mac path for the executable. On other systems, it will be where the `p4merge` binary is installed. Create a merge wrapper script named `extMerge` that calls `p4merge` with all the necessary arguments.

```
$ 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, the wrapper script only passes these arguments.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Make sure these tools are executable.

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Now set up your config file so that Git uses your custom merge and diff tools. Doing so requires 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 the program indicates a successful merge resolution, and `diff.external` to tell Git what command to run to do diffs. So, either run the following 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 add these lines to `~/.gitconfig`:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"
  trustExitCode = false
[diff]
  external = extDiff
```

After all this is done, if you run diff commands such as

```
$ git diff 32d1776b1^ 32d1776b1
```

instead of producing the diff output on the command line, Git fires up P4Merge, which looks something like Figure 7-1.

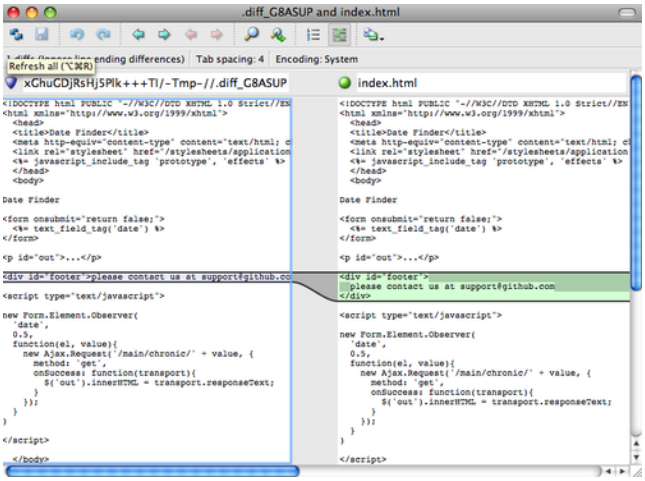

Figure 7.1: P4Merge.

If you try to merge two branches and subsequently have conflicts, run `git mergetool`. It starts P4Merge to the conflicts.

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 merge-resolution tools without you having to do any special 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 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.

7.1.4 Formatting and Whitespace

Formatting and whitespace issues are some of the more frustrating and subtle problems that many developers encounter when collaborating, especially when using multiple platforms. It's very easy for patches or other collaborative work to include subtle text editor introduced whitespace changes or conflicting line-ending markers. Git has a few configuration settings to help with these issues.

core.autocrlf

If any of the people collaborating on a project are using Windows, you'll probably run into line-ending issues at some point. This is because Windows uses both a carriage-return character (CR) and a linefeed character (LF) at the end of lines in files, whereas Mac and Linux systems use only LF. This is a subtle but incredibly annoying fact of cross-platform work.

Git can handle this by automatically converting CRLF line endings into LF when you commit, and vice versa when it checks out code into your working directory. You can turn on this ability 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 do any conversion 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 results in CRLF endings in Windows checkouts but LF endings on Mac and Linux systems and in the Git repository.

If you're a Windows programmer doing a Windows-only project, then you can turn off conversions all together, saving CRLF line endings in the Git repository, by setting the config setting to `false`.

```
$ git config --global core.autocrlf false
```

core.whitespace

Git comes preset to detect and fix four primary whitespace issues — two are enabled by default and can be turned off, and two aren't enabled by default but can be turned on.

The two that are turned on by default are `trailing-space`, which looks for spaces at the end of lines, and `space-before-tab`, which looks for spaces before tabs at the beginning of lines.

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 allowed.

Tell Git which of these you want enabled by setting `core.whitespace` to the settings you want turned on or off, separated by commas. Disable settings by either leaving them out of the setting string or prepending a `-` in front of the setting. For example, if you want all but `cr-at-eol` to be set, run

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git will detect these issues when you run `git diff` and try to warn you by coloring them to give you a chance to fix them before you commit. It will also use these settings to help when you apply patches with `git apply`. When you're applying patches, you can tell 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 `git rebase` as well. If you have commits with whitespace issues but haven't yet pushed upstream, run `git rebase --whitespace=fix` to have Git automatically fix whitespace issues as it's rewriting the patches.

7.1.5 Server Configuration

Not nearly as many configuration settings are available for the server side of Git, but there are a few interesting ones to take note of.

receive.fsckObjects

Although Git can make sure each object still matches its SHA-1 hash and points to valid objects during a push, it doesn't do that by default. This is a relatively expensive operation and may add a lot of time to each push, depending on the size of the repository. For Git to check object consistency on every push, set `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 sending 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, the push will 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 `git push` command.

To disable the ability to force-update remote branches to non-fast-forward references, set `receive.denyNonFastForwards` to true:

```
$ git config --system receive.denyNonFastForwards true
```

The other way to 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 with the new reference. Setting `receive.denyDeletes` to true

```
$ git config --system receive.denyDeletes true
```

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. There are also more interesting ways to do this on a per-user basis via ACLs, as you'll learn at the end of this chapter.

7.2 Git Attributes

Rather than applying to everything in a Git repository, some of these settings can also be specified for a specific path, so that Git applies those settings only for a subdirectory or subset of files. These path-specific settings are called Git attributes and appear either in a `.gitattributes` file in 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, tell Git how to diff non-text files, or have Git filter content before you check it into or out. In this section, you'll learn about some of the attributes you can set and see a few examples of using this feature in practice.

7.2.1 Binary Files

One cool trick for using Git attributes is telling Git which files are binary (in case Git can't figure it out) and giving Git special instructions for handling 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 should be treated as binary. For instance, Xcode projects on the Mac contain a file whose name ends in `.pbxproj`, which is basically a JSON (plain text javascript data format) file created by the Xcode IDE to record your build settings and other values. Although it's technically a text file, because it only contains ASCII characters, 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 makes changes to their own copies, 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 produce a diff showing changes in `.pbxproj` files when you run `git show` or `git diff`. You can also use the built-in macro `binary` that means `-crlf -diff`.

```
*.pbxproj binary
```

Diffing Binary Files

You can use Git attributes to meaningfully diff binary files. You do this by telling Git how to convert your binary data into a text format that can be compared using a normal diff tool.

MS Word files One of the most annoying problems known to humanity is version-controlling Microsoft Word documents. Everyone knows that Word is the most horrific word processor around but, oddly, everyone uses it. 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

```
$ 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 Word files unless you check them out and scan them manually, right? It turns out there's an easy way to do this — using Git attributes. Put the following line in your `.gitattributes` file:

```
*.doc diff=word
```

This tells Git that any file whose name matches this pattern (.doc) should use the "word" filter when to display a diff. What is the "word" filter? You have to create it yourself. Here you configure Git to use the `strings` program to convert Word documents into readable text files, which it can then diff properly.

```
$ git config diff.word.textconv strings
```

This command adds a section to your `.git/config` file that looks like

```
[diff "word"]
textconv = strings
```

Side note: There are different kinds of `.doc` files. Some use an UTF-16 encoding or other "codepages" and `strings` won't find anything useful in there. Your mileage may vary.

Now Git knows that if it tries to do diff two snapshots, and any of the files in the snapshots end in `.doc`, it should run those files through the "word" filter, which is defined as the `strings` program. This effectively makes nice text-based versions of your Word files before attempting to diff them. Because this is a pretty cool and not widely known feature, I'll go over a few examples.

Here's one example. I put a MS Word version of 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
@@ -8,7 +8,8 @@ re going to cover Version Control Systems (VCS) and Git basics
 re going to cover how to get it and set it up for the first time if you don
 t already have it on your system.
 In Chapter Two we will go over basic Git usage - how to use Git for the 80%
-s going on, modify stuff and contribute changes. If the book spontaneously
+s going on, modify stuff and contribute changes. If the book spontaneously
+Let's see if this works.
```

Git successfully and succinctly told me that I added the string “Let’s see if this works”, which is correct. This solution isn’t perfect — it adds a bunch of random stuff at the end — but it certainly works. If you can find or write a Word-to-plain-text converter that works better, then the Git community will be grateful. However, `strings` is available on most Mac and Linux systems, so this approach may be a good first try.

OpenDocument Text files The same approach that I 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, add the `odt` diff filter in `.git/config`.

```
[diff "odt"]
binary = true
textconv = /usr/local/bin/odt-to-txt
```

OpenDocument files are actually zipped directories containing multiple files (the content in XML format, stylesheets, images, etc.). You’ll need to write a script to extract the content as plain text. Create the file `/usr/local/bin/odt-to-txt` (it can go in any 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 diff PNG format files is to run them through a filter that extracts their EXIF information — metadata that's recorded with most image formats. If you install the `exiftool` program, you can use it to see the metadata in text format, so at least the diff will show a textual representation of any changes.

```
$ 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

```
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.

7.2.2 Keyword Expansion

SVN- or CVS-style keyword expansion is an often requested feature. The main problem with this in Git is that you can't modify a file with information about a 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 staged. Git attributes offer two ways to do this.

First, you can inject the SHA-1 hash of a blob into an `$Id$` field in the file automatically. If you set this attribute on a file, then the next time you check out the file, Git will replace that field with the SHA-1 hash of the blob. It's important to notice that it isn't the SHA-1 hash 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-1 hash 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-1 hash isn't all that helpful, because it's completely random so you can't tell if one SHA-1 hash is older or newer than another.

It turns out that you can write your own filters for doing substitutions in files when they're committed or checked out. These are the "clean" and "smudge" filters. In the `.gitattributes` file, set a filter for particular paths and then create scripts that process files just before they're checked out ("smudge", see Figure 7-2) or just before they're committed ("clean", see Figure 7-3). These filters can be set up to do all sorts of fun things.

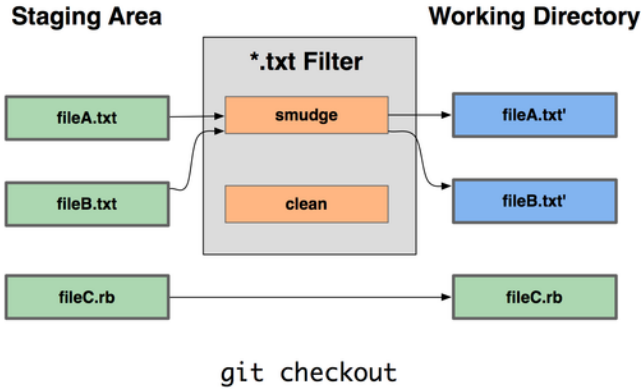

Figure 7.2: The "smudge" filter is run on checkout.

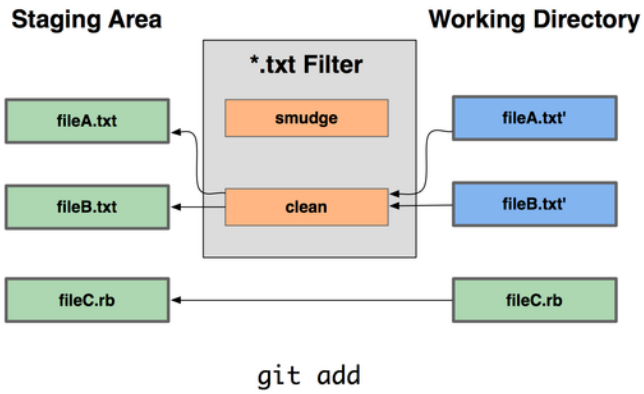

Figure 7.3: The "clean" filter is run when files are staged.

The original commit message for this feature shows 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 `.gitattributes` 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 whose name matches `*.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. 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 does `$Date$` 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 containing the file, and then inserts the date into the file. Here's 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('$Date$', '$Date: ' + last_date.to_s + '$')
```

The script gets the latest commit date from the `git log` command, sticks the date into any `$Date$` string it finds in stdin, and prints the results. This should be simple to do in whatever language you're most comfortable in. You can name this script `expand_date` and put it in your path. Now, you need to set up a filter (call it `dater`) and tell Git 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, test it by creating a file containing a `$Date$` keyword and then setting up a Git attribute for that file that invokes 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, this won't work everywhere. When you design these filters, they should be able to fail without damaging the project.

7.2.3 Exporting Your Repository

Git attributes also allow some interesting things to happen 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's anything that you don't want to include in an archive but that you do want checked into your project, configure the `export-ignore` attribute to bypass what you don't want included.

For example, say you have some test files in a `test/` subdirectory, and it doesn't make sense to include them in the export of your project. 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

You can also do some simple keyword substitution in your archives. 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, to include a file named `LAST_COMMIT` containing a project's last commit date, with the last commit date automatically injected into it when you run `git archive`, set up `LAST_COMMIT` to look like

```
$ 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 `LAST_COMMIT` in the archive file look like

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

7.2.4 Merge Strategies

You can also use Git attributes to select different merge strategies for specific files in your project. One very useful setting is for 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, but you want to be able to merge changes back in from it, ignoring certain files. Say you have a database settings file called `database.xml` that's different in two branches, and you want to merge in your other branch without messing up the file. Set up an attribute like

```
database.xml merge=ours
```

If you merge the other branch, instead of having merge conflicts with `database.xml`, you see something like

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In this case, `database.xml` remains unchanged.

7.3 Git Hooks

Like many other VCSs, Git can fire off custom scripts when certain important actions occur. These are called hooks. There are two groups of 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 server operations such as receiving pushed commits. You can use these hooks for all sorts of purposes. You'll learn about a few of them here.

7.3.1 Installing a Hook

The hooks are all stored in the `.git/hooks` subdirectory. By default, Git populates this directory with a bunch of example scripts, many of which are useful by themselves. But the examples also document the input values of each script. Most of the examples are written as shell scripts, but there's some Perl thrown in. Any properly named executable script will work fine — you can write them in Ruby, Python, or what have you. The example hook files end with `.sample` — you'll need to rename them in order for them to be executed. I'll cover most of the major hook filenames here.

7.3.2 Client-Side Hooks

There are a lot of client-side hooks. This section splits them into committing-workflow scripts, e-mail-workflow scripts, and everything else.

Committing-Workflow Hooks

The first four hooks have to do with the commit process. Git runs the `pre-commit` hook first, before you even enter a commit message. This hook can be used to inspect the snapshot that's about to be committed to see if you've forgotten something, to make sure tests have run, or to examine whatever you need to inspect in the code. A non-zero exit status from this hook aborts the commit, although you can override this check with the `--no-verify` option to `git commit`. Some other things you can do are 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.

Git runs the `prepare-commit-msg` hook before the commit message editor is fired up but after the default message is created. It lets you edit the default message before the committer 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 hash 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 commit messages generated from templates, merge commits, squashed commits, and amended commits. You may use it in conjunction with a commit template to programmatically insert text into the commit message.

The `commit-msg` hook takes one parameter, which is the path to a temporary file that contains the current commit message. If this script exits with a non-zero value, Git aborts the commit process. This hook is useful for validating your project state or commit message before allowing a commit to take place. In the last section of this chapter, I'll demonstrate using this hook to check that your commit message conforms 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 SHA-1 hash of the last commit by running `git log -1 HEAD`. Generally, this script is used for notifications 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 a 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. These scripts help developers, and they must be set up and maintained by them, although the scripts can be overridden or modified 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 accept patches over e-mail prepared by `git format-patch`, then some of these hooks may be helpful.

The first hook that Git runs 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 with a non-zero value. Use this hook to make sure a commit message is properly formatted or to modify the message by having the script edit it in place.

The next hook that runs 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 committing. You can run tests or otherwise inspect the working directory with this

script. If something is missing or the tests don't pass, exiting with a non-zero value also aborts the `git am` command without committing the patch.

The last hook that runs during a `git am` operation is `post-applypatch`. You can use it to notify a group or the author of the patch that you've applied the patch. 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 with a non-zero value. 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 the name of your stable published branch.

After you run a successful `git checkout`, the `post-checkout` hook runs. 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 `git merge` command. Use it to restore data in the working tree that Git can't track, such as permissions data. This hook can likewise validate files out of Git control that you may want copied in when the working tree changes.

7.3.3 Server-Side Hooks

In addition to the client-side hooks, you can use a couple of important server-side hooks to enforce nearly any kind of policy. These scripts run before and after pushes to the server. The pre hooks can exit with non-zero values at any time to reject the push as well as send 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 that runs when handling a push from a client is `pre-receive`. It takes a list from stdin of the references that are being pushed. If it exits with a non-zero value, none of the pushes are accepted. 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.

The `post-receive` hook runs after the entire push process is complete and can be used to update other services or notify users. It takes the same data from stdin 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 the push has completed. So be careful using this hook to do anything that may take a long time.

update

The `update` script is very similar to the `pre-receive` script, except that `update` is run once for each branch the push is trying to update. If the push goes to multiple branches, `pre-receive` runs only once, whereas `update` runs once for each branch being pushed to. Instead of reading from stdin, this script takes three arguments — the name of the reference (branch), the SHA-1 hash that reference pointed to before the push, and the SHA-1 hash the user is trying to push. If the update script exits with a non-zero value, only that reference is rejected. Other references can still be updated.

7.4 An Example Git-Enforced Policy

In this section, you'll use what you've learned to establish a Git workflow that checks for a prescribed commit message format, enforces fast-forward-only pushes, and allows only certain users to modify certain subdirectories in a project. You'll build client scripts that tell the developer if their push will be rejected, and server scripts that actually enforce the policies.

I used Ruby to write the scripts, both because it's my preferred scripting language and because I feel it's the most pseudocode-looking of the scripting languages. Thus, you should be able to roughly follow the code even if you don't use Ruby. However, any language will work fine. All the sample hook scripts distributed with Git are written in either Perl or Bash, so you can also see plenty of examples of hooks in those languages by looking at the samples.

7.4.1 Server-Side Hook

All the server-side work will go into the `update` script in your hooks directory. This script runs once per branch being pushed and takes the reference being pushed to, the old revision where that branch was, and the new revision being pushed. You can also retrieve the username of the user doing the pushing if the push is being sent over SSH. If you've allowed everyone to connect as the same user (like "git") via public-key authentication, you may have to create a shell wrapper for that user that determines which remote user is connecting based on the public key, and set an environment variable containing that username. Here I assume the name of the connecting user is in the `$USER` environment variable, so the `update` script begins by gathering all the information it needs.

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies... \n(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

Yes, I'm using global variables. Don't judge me — it's easier to demonstrate in this manner.

Enforcing a Specific Commit-Message Format

Your first challenge is to enforce the policy that each commit message must adhere to a particular format. As a sample specification, assume that each message has to include a string that looks like "ref: 1234" because you want each commit to link to a specific ticket in your ticketing system. You must look at each commit being pushed, check if a string in that format is in the commit message, and, if not, exit with a non-zero value so the push is rejected.

You can get a list of the SHA-1 hashes of all the commits that are being pushed by passing the value of `$newrev` and `$oldrev` to a Git plumbing command called `git rev-list`. This is similar to the `git log` command, but by default it outputs only SHA-1 hashes and nothing else. So, to get a list of all the commit SHA-1 hashes introduced between one commit's SHA-1 hash and another, run something like

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Take that output, loop through each of those commit SHA-1 hashes, grab the commit message for each commit, and test that message against a regular expression that looks for the required pattern.

Next, get the commit message from each of these commits in order to test them. To get the complete commit data, use the `git cat-file` plumbing command. I'll go over all these plumbing commands in detail in Chapter 9 but for now, here's what that command gives you.

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

A simple way to get the commit message when you have the commit's SHA-1 hash is to go to the first blank line and take everything after that. The `sed` command on Unix systems can do this.

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
changed the version number
```

Use that incantation to select the commit message from each commit in the push and exit if anything doesn't match. To exit the script and reject the push, exit with a non-zero value. The whole method looks like this:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

Putting that in your `update` script will reject updates that contain commits with messages that don't adhere to your rule.

Enforcing a User-Based ACL System

Suppose you want to add a mechanism that uses an access control list (ACL) that specifies which users are allowed to push changes to certain parts of your projects. Some users have full access, and others only have access to push changes to certain subdirectories or specific files. To implement this, put those rules in the file `~/.git/acl`. The `update` hook looks at those rules, sees what files are in all the commits being pushed, and determines whether the user doing the push has permission to update all those files.

The first thing to do is create the ACL. Here you'll use a format very much like the CVS ACL syntax. It uses a series of lines, where the first field is `avail` or `unavail`, the next field is a comma-delimited list of the users to which the rule applies, and the last field is the path to which the rule applies (blank means open access). All of these fields are delimited by a vertical bar (`|`) character.

In this case, there are a couple of administrators, some documentation writers with access to the `doc` directory, and one developer who only has access to the `lib` and `tests` directories. Your ACL file looks like

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Begin by reading this data into a structure. In this case, to keep the example simple, I only enforce the `avail` directives. Here's a method that contains an associative array where the key is the user name and the value is an array of paths to which the user has write access.

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

In the ACL file you looked at earlier, the `get_acl_access_data` method returns a data structure that looks like

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Now that you have the permissions sorted out, determine what paths the commits being pushed modify to make sure the user doing the pushing has access to all of them.

You can pretty easily see what files have been modified in a single commit with the `--name-only` option to the `git log` command (mentioned briefly in Chapter 2).

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

If you use the ACL structure returned from the `get_acl_access_data` method and check it against the files listed in each of the commits, you can determine whether the user has access to push all of their commits.

```
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --
pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path || # user has access to everything
          (path.index(access_path) == 0) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

Most of that should be easy to follow. Get a list of new commits being pushed to your server with `git rev-list`. Then, for each of those commits, find which files are modified and make sure the user who's doing the push has access to all the paths being modified. One Rubyism that may not be clear is `path.index(access_path) == 0`, which is true if path begins with `access_path`. This ensures that `access_path` is not just in one of the allowed paths, but appears at the beginning of each accessed path.

Now your users can't push any commits with badly formed messages or with modified files outside of their designated paths.

Enforcing Fast-Forward-Only Pushes

The only thing left is to set the `receive.denyDeletes` and `receive.denyNonFastForwards` settings to enforce fast-forward-only pushes. Enforcing this with a hook will also work, and you can modify the hook to apply only for certain users or whatever else you come up with later.

The logic for this script is to see if any commits are reachable from the older revision that aren't reachable from the newer one. If there are none, then it was a fast-forward push. Otherwise, deny the push.

```
# enforces fast-forward only pushes
def check_fast_forward
  missed_refs = `git rev-list #{$newrev}..#{$oldrev}`
  missed_ref_count = missed_refs.split("\n").size
  if missed_ref_count > 0
    puts "[POLICY] Cannot push a non fast-forward reference"
    exit 1
  end
end

check_fast_forward
```

Everything is ready. If you run `chmod u+x .git/hooks/update`, which is the file which should contain all this code, and then try to push a non-fast-forward reference, you'll get something like

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Cannot push a non fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

There are a couple of interesting things here. When the hook starts running you see

```
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
```

You sent that to stdout at the very beginning of your update script. It's important to note that anything your script sends to stdout will be received by the client.

The next thing you'll notice is the error message.

```
[POLICY] Cannot push a non fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

The first line was printed by your script. The other two lines are from Git saying that the update script exited with a non-zero value, and that's what canceled the push. Lastly, you'll see

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

You'll see a remote rejected message for each reference that your hook declined, saying that the push was declined specifically because of a hook failure.

Furthermore, if the ref marker doesn't appear in any of your commits, you'll see an error message.

```
[POLICY] Your message is not formatted correctly
```

Or, if someone tries to edit a file they don't have access to and then push a commit containing the file, the result will look similar. For instance, if a documentation author tries to push a commit modifying something in the `lib` directory, they see

```
[POLICY] You do not have access to push to lib/test.rb
```

That's all. From now on, as long as that `update` script is executable, your repository will never be rewound, will never contain a commit message that doesn't conform to your requirements, and your users will be unable to change files to which they've been refused access.

7.4.2 Client-Side Hooks

The downside to this approach is the whining that will inevitably result when users' pushes are rejected. Having carefully crafted work rejected at the last minute can be extremely frustrating and confusing. Furthermore, they will have to edit their commit history to correct the violation, which isn't always for the faint of heart.

The answer to this dilemma is to provide client-side hooks that users can use to be notified when they do something that the server is likely to reject. That way, they can correct any problems before committing and before those issues become more difficult to fix. Because hooks aren't transferred when a project is cloned, you must distribute these scripts some other way and then have your users copy them to their `.git/hooks` directory and make them executable. You can distribute these hooks within the project or in a separate project, but there's no way to set them up automatically.

To begin, check your commit message just before each commit is recorded, so you know the server won't reject your change due to a badly formatted commit message. To do this, add the `commit-msg` hook. It reads the message from the file passed as the first argument and compares the contents of the file that to the required pattern. Git aborts the commit if there's no match.

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

If that script is in place (in `.git/hooks/commit-msg`) and executable, and you commit with a message that isn't properly formatted, you see

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

No commit was completed. However, if your message contains the proper pattern, Git allows the commit.

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 file changed, 1 insertion(+), 0 deletions(-)
```

Next, make sure you aren't modifying files that are outside your ACL scope. If your project's `.git` directory contains a copy of the ACL file you used previously, then the following `pre-commit` script will enforce this constraint:

```
#!/usr/bin/env ruby

$user    = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

This is roughly the same script as the server-side version, but with two important differences. First, the ACL file is in a different place because this script runs from your working directory, not from your Git directory. You have to change the path to the ACL file from this

```
access = get_acl_access_data('acl')
```

to this.

```
access = get_acl_access_data('.git/acl')
```

The other important difference is the way you get a listing of the files that have changed. Because the server-side method looks at the log of commits, and, at this point, the commit hasn't been recorded yet, you must get your file listing from the staging area instead. Instead of

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

use

```
files_modified = `git diff-index --cached --name-only HEAD`
```

But, those are the only differences. Otherwise, the script works the same way. One caveat is that the script expects you to be running locally as the same user you connect as on the remote machine. If that's not true, set the `$user` variable manually.

The last thing to do is check that you're not trying to push non-fast-forwarded references, but that's a bit less common. To get a non-fast-forward reference, you either have to rebase past a commit you've already pushed or try pushing a different local branch to the same remote branch.

Because the server will tell you that you can't push a non-fast-forward reference anyway, and the hook prevents forced pushes, the only thing you can try to catch is accidental rebasing commits that have already been pushed.

Here's an example pre-rebase script that checks for that. It gets a list of all the commits you're about to rewrite and checks whether they exist in any of your remote references. If it sees one that's reachable from one of your remote references, it aborts the rebase.

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
```

```
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
endG
```

This script uses a syntax that wasn't covered in the Revision Selection section of Chapter 6. Get a list of commits that have already been pushed by running

```
git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}
```

The `SHA^@` syntax resolves to all the parents of that commit. You're looking for any commit that's reachable from the last commit on the remote that isn't reachable from any parent of any of the SHA-1 hashes you're trying to push — meaning it's a fast-forward commit.

The main drawback to this approach is that it can be very slow and is often unnecessary. If you don't try to force the push with `-f`, the server will warn you and not accept the push anyway. However, it's an interesting exercise and can, in theory, help avoid a rebase that you might later have to go back to fix.

7.5 Summary

I've covered most of the major ways you can customize your Git client and server to best fit your workflows. You've learned about all sorts of configuration settings, file-based attributes, and event hooks, and you've built an example policy-enforcing script. You should now be able to make Git fit nearly any workflow you can dream up.

Chapter 8

Git and Other Systems

The world isn't perfect. Usually, you can't immediately switch every project you come in contact with to Git. Sometimes you're stuck working on a project using another VCS, and many times that system is Subversion. The first part of this chapter covers learning about `git svn`, Git's bidirectional Subversion gateway tool. At some point, you may want to convert your existing Subversion project to Git. The second part of this chapter covers how to migrate your project to Git — first from Subversion, then from Perforce, and finally via a custom import script for nonstandard imports.

8.1 Git and Subversion

Currently, the majority of open source development projects and a large number of corporate projects use Subversion to manage their source code. It's the most popular open source VCS and has been around for nearly a decade. It's also very similar in many ways to CVS, which was the big boy of the source-control world before Subversion came along.

One of Git's great features is a bidirectional bridge to Subversion called `git svn`. This tool allows using Git as a Subversion client so you can use all the local features of Git and then push to a Subversion server. This means you can do local branching and merging, use the staging area, rebasing, and cherry-picking, and so on, while your collaborators continue to work in their dark and ancient ways. It's a good way to sneak Git into the corporate environment and help your fellow developers become more efficient while you lobby to get the infrastructure changed to fully support Git. The Subversion bridge is the gateway drug to the DVCS world.

8.1.1 git svn

The command in Git for all the Subversion bridge commands is `git svn`. Preface everything with that. It takes quite a few options, so I'll present the common ones while going through a few small workflows.

It's important to note that when you're using `git svn`, you're interacting with Subversion, which is a system that's far less sophisticated than Git. Although you can easily do local branching and merging, it's generally best to keep your history as linear as possible by rebasing your work and avoiding doing things like simultaneously interacting with a Git remote repository.

Don't rewrite your history and try to push again, and don't push to a parallel Git repository to collaborate with fellow Git developers at the same time. Subversion can have only a single linear history, and confusing it is very easy. If you're working with a team, and some members are using Subversion and others are using Git, make sure everyone is using the Subversion server to collaborate. Doing so will make your life easier.

8.1.2 Setting Up

To demonstrate the Subversion bridge, you need a Subversion repository that you have write access to. To copy these examples, you'll have to make a writeable copy of my test repository. In order to do that easily, use the `svnsync` tool that comes with Subversion 1.4 and later. For these tests, I created a new Subversion repository on Google code that was a partial copy of the `protobuf` project, which is a tool that encodes structured data for network transmission.

To follow along, first create a new local Subversion repository.

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Then, enable all users to change revprops. The easy way is to add a pre-revprop-change script that always exits with the value 0.

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Now sync this project to your local machine by running `svnsync init` with the to and from repositories as parameters.

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

This sets up the properties to run the sync. You can then clone the project by running

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
...
```

Although this operation may only take a few minutes, if you try to copy the original repository to another remote repository instead of a local one, the process will take nearly an hour, even though there are fewer than 100 commits. Subversion has to clone one revision at a time and then push the revision back into another repository. It's ridiculously inefficient, but it's the only easy way to do this.

8.1.3 Getting Started

Now that you have a Subversion repository to which you have write access, go through a typical workflow. Start with the `git svn clone` command, which imports an entire Subversion repository into a local Git repository. Remember that if you're importing from a real hosted Subversion repository, you should replace `file:///tmp/test-svn` with the URL of your Subversion repository.

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/
svn/.git/
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
      A    m4/acx_pthread.m4
      A    m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
    file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
 file:///tmp/test-svn/branches/my-calc-branch r76
```

This runs the equivalent of two commands — `git svn init` followed by `git svn fetch` — on the URL you provide. This can take a while. The test project has only about 75 commits and the codebase isn't that big, so it takes just a few minutes. However, Git has to check out each version, one at a time, and commit it individually. For a project with hundreds or thousands of commits, this can literally take hours or even days to finish.

The `-T trunk -b branches -t tags` parameters tell Git that this Subversion repository follows the basic branching and tagging conventions. If you name your trunk, branches, or tags differently, you can change these options. Because this is so common, you can replace this entire sequence with `-s`, which means to use a standard layout and implies all those options. The following command is equivalent to the `git svn clone` command above:

```
$ git svn clone file:///tmp/test-svn -s
```

At this point, you should have a valid Git repository containing your imported branches and tags.

```
$ git branch -a
* master
  my-calc-branch
  tags/2.0.2
  tags/release-2.0.1
  tags/release-2.0.2
  tags/release-2.0.2rc1
  trunk
```

It's important to note that this tool namespaces your remote references differently than what you might be used to. When you're cloning a normal Git repository, all the branches on that remote server are available locally with names something like `origin/[branch]` - namespaced by the name of the remote. However, `git svn` assumes that you won't have multiple remotes and saves all its references to points on the remote server with no namespacing. Use the Git plumbing command `git show-ref` to see full reference names.

```
$ git show-ref
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk
```

A normal Git repository looks more like

```
$ git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

You have two remote servers, one named `gitserver` with a `master` branch and another named `origin` with two branches, `master` and `testing`.

Notice how in the example of remote references imported using `git svn`, tags are added as remote branches, not as real Git tags. Your Subversion import looks like it has a remote named `tags` with branches under it.

8.1.4 Committing Back to Subversion

Now that you have a working repository, do some work on the project and push your commits back upstream, effectively using Git as an Subversion client. Any commits only exist in Git locally and not on the Subversion server.

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 97031e5] Adding git-svn instructions to the README
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Next, push your change upstream. Notice how this is different from the way you normally use Subversion. You can do several commits offline and then push them all at once to the Subversion server. To push to a Subversion server, run `git svn dcommit`.

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      README.txt
Committed r79
       M      README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

This does a Subversion commit for each commit you've made, and then rewrites your local Git commit to include a unique identifier. This is important because it means that all the SHA-1 hashes for your commits change. This is one reason why working with Git-based remote versions of your projects concurrently with a Subversion server isn't a good idea. If you look at the last commit, you see the new `git-svn-id` that was added.

```
$ git log -1
commit 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sat May 2 22:06:44 2009 +0000

    Adding git-svn instructions to the README

             git-svn-id:   file:///tmp/test-svn/trunk@79   4c93b258-373f-11de-
be05-5f7a86268029
```

Notice that the SHA-1 hash that originally started with `97031e5` when you committed now begins with `938b1a5`. To push to both a Git server and a Subversion server, you have to push (`dcommit`) to the Subversion server first, because that changes your commit data.

8.1.5 Pulling New Changes

If you're working with other developers at some point you'll face merge conflicts. A proposed commit will be rejected until you merge the conflicted files. With `git svn`, it looks like

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
Merge conflict during commit: Your file or directory 'README.txt' is probably \
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
core/git-svn line 482
```

To resolve this, run `git svn rebase`, which pulls down any changes on the server that you don't have yet and rebases any work you have on top of what is on the server.

```
$ git svn rebase
       M      README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

Now, all your work is on top of what's on the Subversion server, so you can successfully `dcommit`.

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      README.txt
Committed r81
       M      README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

It's important to remember that unlike Git, which requires merging upstream work you don't have in your working directory before you can push, `git svn` makes you do that only if the changes conflict. If someone pushes a change to one file and then you push a change to another file, `git svn dcommit` will work fine.

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
```

```
       M      configure.ac
Committed r84
       M      autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
       M      configure.ac
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
  using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
  015e4c98c482f0fa71e4d5434338014530b37fa6 M   autogen.sh
First, rewinding head to replay your work on top of it...
Nothing to do.
```

This is important to remember because the outcome is a project state that didn't exist on either computer when you pushed. If the changes are incompatible but don't conflict, you may get issues that are difficult to diagnose. This is different than when using a Git server. In Git, you can fully test the project state on your client system before publishing it whereas with Subversion you can never be certain that the states immediately before and after committing are identical.

You should also run this command to pull changes from the Subversion server even if you're not ready to commit. You can run `git svn fetch` to grab the new data, but `git svn rebase` does the fetch and then updates your local commits.

```
$ git svn rebase
       M      generate_descriptor_proto.sh
r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/trunk.
```

Running `git svn rebase` every once in a while insures your code is always up to date. Be sure your working directory is clean when you run this, though. If you have local changes, you must either stash your work or temporarily commit it before running `git svn rebase`. Otherwise, the command will stop if it sees that the rebase will result in a merge conflict.

8.1.6 Git Branching Issues

When you've become comfortable with a Git workflow, you'll likely create topic branches, do work on them, and then merge them. If you're pushing to a Subversion server via `git svn`, you may want to rebase your work onto a single branch each time instead of merging branches together. The reason to prefer rebasing is that Subversion has a linear history and doesn't deal with merges like Git does, so `git svn` follows only the first parent when converting the snapshots into Subversion commits.

Suppose your history looks like the following: you created an `experiment` branch, did two commits, and then merged them back into `master`. When you `dcommit`, you see output like

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      CHANGES.txt
Committed r85
       M      CHANGES.txt
r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
COPYING.txt: locally modified
INSTALL.txt: locally modified
       M      COPYING.txt
       M      INSTALL.txt
Committed r86
       M      INSTALL.txt
       M      COPYING.txt
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Running `dcommit` on a branch with merged history works fine, except that when you look at your Git project history, it hasn't rewritten either of the commits you made on the `experiment` branch. Instead, all those changes appear in the Subversion version of the single merge commit.

When someone else clones that repository, all they see is the merge commit with all the work squashed into it. They don't see the commit data showing where it came from or when it was committed.

8.1.7 Subversion Branching

Branching in Subversion isn't the same as branching in Git. If you can avoid branching in Subversion, that's probably best. However, you can create and commit to branches in Subversion using `git svn`.

Creating a New Subversion Branch

To create a new branch in Subversion, run `git svn branch [branchname]`.

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/
opera...
```

```
Found possible branch point: file:///tmp/test-svn/trunk => \
  file:///tmp/test-svn/branches/opera, 87
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
Following parent with do_switch
Successfully followed parent
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

This does the equivalent of `svn copy trunk branches/opera` in Subversion and operates on the Subversion server. It's important to note that it doesn't change your current branch. If you commit at this point, that commit will go to `trunk` on the server, not `opera`.

8.1.8 Switching Active Branches

Git figures out what branch your dcommits go to by looking for the tip of any of your Subversion branches in your history. You should have only one, and it should be the last one with a `git-svn-id` in your current branch history.

To work on more than one branch simultaneously, set up local branches to dcommit to specific Subversion branches by starting them at the imported Subversion commit for that branch. If you want an `opera` branch that you can work on separately, run

```
$ git branch opera remotes/opera
```

Now, if you want to merge your `opera` branch into `trunk` (your `master` branch), do a normal `git merge`. But you need to provide a descriptive commit message (via `-m`), or the merge will say "Merge branch opera" instead of something useful.

Remember that although you're using `git merge` to do this operation, and the merge likely will be much easier than it would be in Subversion (because Git will automatically detect the appropriate merge base), this isn't a normal Git merge commit. You have to push this data back to a Subversion server that can't handle a commit that tracks more than one parent. So, after you push it, it will look like a single commit that squashed in all the work of another branch under a single commit. After you merge one branch into another, you can't easily go back and continue working on that branch, as you normally can in Git. The `git svn dcommit` command erases any information that says what branch was merged in, so subsequent merge-base calculations will be wrong. The dcommit makes your `git merge` result look like you ran `git merge --squash`. Unfortunately, there's no good way to avoid this. Subversion can't store this information, so you'll always be crippled by its limitations while using it as your server. To avoid these issues, you should delete the local branch (in this case, `opera`) after you merge it into `trunk`.

8.1.9 Subversion Commands

The `git svn` toolset provides a number of commands to help ease the transition to Git by providing some capabilities that are similar to what you had in Subversion. Here are a few such commands.

Subversion Style History

If you're used to Subversion and want to see your history in Subversion output style, run `git svn log`.

```
$ git svn log
------------------------------------------------------------------------
r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines

autogen change

------------------------------------------------------------------------
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines

Merge branch 'experiment'

------------------------------------------------------------------------
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines

updated the changelog
```

You should know two important things about `git svn log`. First, it works offline, unlike the real `svn log` command, which asks the Subversion server for the data. Second, it only shows commits that have been made to the Subversion server. Local Git commits that you haven't dcommited don't show up. Neither do commits that others have made to the Subversion server in the meantime. It's more like seeing the last known state of the commits on the Subversion server.

Subversion Annotation

Much as the `git svn log` command simulates the `svn log` command offline, you can get the equivalent of `svn annotate` by running `git svn blame [FILE]`. The output looks like

```
$ git svn blame README.txt
 2   temporal Protocol Buffers - Google's data interchange format
 2   temporal Copyright 2008 Google Inc.
 2   temporal http://code.google.com/apis/protocolbuffers/
 2   temporal
22   temporal C++ Installation - Unix
22   temporal =======================
 2   temporal
79    schacon Committing in git-svn.
78    schacon
 2  temporal To build and install the C++ Protocol Buffer runtime and the Protocol
```

```
 2   temporal Buffer compiler (protoc) execute the following:
 2   temporal
```

Again, it doesn't show commits that you did locally in Git or that have been pushed to Subversion in the meantime.

Subversion Server Information

You can also get the same sort of information that `svn info` shows by running `git svn info`.

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

This is like `git blame` and `git log` in that it runs offline and is up to date only as of the last time you communicated with the Subversion server.

Ignoring What Subversion Ignores

If you clone a Subversion repository that has `svn:ignore` properties set anywhere, you'll likely want to create corresponding `.gitignore` files so you don't accidentally commit files that you shouldn't. `git svn` has two options to help with this. The first is `git svn create-ignore`, which automatically creates corresponding `.gitignore` files so your next commit can include them.

The second option is `git svn show-ignore`, which outputs the lines you need to put in a `.gitignore` file. So, redirect the output into your project exclude file.

```
$ git svn show-ignore > .git/info/exclude
```

That way, you don't litter the project with `.gitignore` files. This is a good option if you're the only Git user on a Subversion team, and your teammates don't want `.gitignore` files in the project.

8.1.10 Git-Svn Summary

The `git svn` tools are useful if you're stuck in a development environment that must run a Subversion server. You should consider it as a crippled Git, however, or you'll hit confusing translation issues. To stay out of trouble, try to follow the following guidelines:

- Keep a linear Git history that doesn't contain merge commits made by `git merge`. Rebase any work you do outside your mainline branch back onto it. Don't merge the work in.
- Don't establish a separate Git server. It might be worthwhile having one to speed up cloning for new developers, but don't push anything to it that doesn't have a `git-svn-id` entry. You may even want to add a `pre-receive` hook that checks each commit message for a `git-svn-id` and rejects pushes that contain commits without it.

If you follow those guidelines, working with a Subversion server can be more bearable. However, moving to a real Git server will result in immediate benefits.

8.2 Migrating to Git

If you have an existing project in another VCS but you've decided to switch to Git, you must somehow migrate your project. This section goes over some importers that are included with Git for common systems and then demonstrates how to develop your own custom importer.

8.2.1 Importing

You'll learn how to import data from two of the most commonly used VCSs — Subversion and Perforce — both because their users make up the majority of users I hear of who are currently switching VCSs, and because high-quality tools for importing from both systems are distributed with Git.

8.2.2 Subversion

If you read the previous section about using `git svn`, you can easily use those instructions to `git svn clone` a repository. Then, stop using the Subversion server, push to a new Git server, and start using that. If you want the change history, you can accomplish that as quickly as you can pull the data out of the Subversion server (which may take a while).

However, the import isn't perfect and because it will take so long, you may as well do it right. The first problem is importing author information. In Subversion, the username of each person committing is recorded in the commit information. The examples in the previous section show `schacon` in some places, such as the `blame` and the `git svn log` output. To map this to a full email address, which is the convention for Git author information, you need a mapping from Subversion users to Git authors. Create a file called `users.txt` containing this mapping in this format.

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

To get a list of the author names that Subversion uses, run

```
$ svn log --xml | grep -P "^<author" | sort -u | \
      perl -pe 's/<author>(.*?)<\/author>/$1 = /' > users.txt
```

That produces output in XML format. Look for the authors, create a unique list, and then strip out the XML. Then, redirect that output into `users.txt` to add the equivalent Git author information next to each entry.

You can pass this file to `git svn` to help map the author information more accurately. You can also tell `git svn` not to include the metadata that Subversion normally exports by adding `--no-metadata` to the `git svn clone` or `git svn init` command. This makes your import command look like

```
$ git svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata -s my_project
```

Now you should have a nice Subversion import in your `my_project` directory. Instead of commits that look like

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

                        git-svn-id:      https://my-project.googlecode.com/svn/
trunk@94 4c93b258-373f-11de-
    be05-5f7a86268029
```

they look like

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Not only does the Author field look a lot better, but `git-svn-id` is no longer there, either.

You need to do a bit of post-import cleanup. For one thing, you should clean up the weird references that `git svn` created. First, change the tags so they're actual tags rather than strange remote branches, and then change the rest of the branches so they're local.

To change the tags into proper Git tags, run

```
$ git for-each-ref refs/remotes/tags | cut -d / -f 4- | grep -
v @ | while read tagname; do git tag "$tagname" "tags/$tagname"; git branch -
r -d "tags/$tagname"; done
```

This takes the references that were remote branches that started with `tag/` and makes them real (lightweight) tags.

Next, change the rest of the references under `refs/remotes` to be local branches.

```
$ git for-each-ref refs/remotes | cut -d / -f 3- | grep -
v @ | while read branchname; do git branch "$branchname" "refs/remotes/
$branchname"; git branch -r -d "$branchname"; done
```

Now all the old branches are real Git branches and all the old tags are real Git tags. The last thing to do is add your new Git server as a remote and push to it like

```
$ git remote add origin git@my-git-server:myrepository.git
```

Because you want all your branches and tags to go to your new Git server, now run

```
$ git push origin --all
$ git push origin --tags
```

8.2.3 Perforce

The next system you'll look at importing from is Perforce. A Perforce importer is also distributed with Git. In versions of Git before 1.7.11, the importer is only available in the `contrib` section of the source code. In that case you must get the Git source code, which you can download from git.kernel.org.

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/contrib/fast-import
```

In the `fast-import` directory, you should find an executable Python script named `git-p4`. You must have Python and the `p4` tool installed on your machine for this script to work. For example, you'll need to import the Jam project from the Perforce Public Depot. To set up your client, you must set the P4PORT environment variable to point to the Perforce depot.

```
$ export P4PORT=public.perforce.com:1666
```

Run `git-p4 clone` to import the Jam project from the Perforce server, supplying the depot, project path, and the path to contain the project.

```
$ git-p4 clone //public/jam/src@all /opt/p4import
Importing from //public/jam/src@all into /opt/p4import
Reinitialized existing Git repository in /opt/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 4409 (100%)
```

If you go to the `/opt/p4import` directory and run `git log`, you can see your imported work.

```
$ git log -2
commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
Author: Perforce staff <support@perforce.com>
Date:   Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

    [git-p4: depot-paths = "//public/jam/src/": change = 4409]

commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date:   Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c

    [git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

Notice the `git-p4` identifier in each commit. It's fine to keep that identifier there, in case you need to reference the Perforce change number later. However, if you'd like to remove

it, now's the time to do so — before you start doing work on the new repository. Use `git filter-branch` to remove the identifier strings en masse.

```
$ git filter-branch --msg-filter '
        sed -e "/^\[git-p4:/d"
'
Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

If you run `git log`, you see that all the SHA-1 hashes for the commits have changed, but the `git-p4` strings are no longer in the commit messages.

```
$ git log -2
commit 10a16d60cffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date:   Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rmg@perforce.com>
Date:   Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c
```

Your import is ready to push to your new Git server.

8.2.4 A Custom Importer

If your VCS isn't Subversion or Perforce, you should look for an importer online. Quality importers are available for CVS, Clear Case, Visual Source Safe, and even a directory of archives. If none of these tools work for you, you have some other VCS, or you otherwise need a more custom importing process, use `git fast-import`. This command reads simple instructions from stdin to write specific Git data. It's much easier to create Git objects this way than to run the raw Git commands or try to write the raw objects (see Chapter 9 for more information). This way, you can write an import script that reads the necessary information from the system you're importing from and sends straightforward instructions to stdout. You can then run this program and pipe its output through `git fast-import`.

To quickly demonstrate, let's write a simple importer. Suppose you work in `current`, you back up your project by occasionally copying the directory into a time-stamped `back_YYYY_MM_DD` backup directory, and you want to import this into Git. Your directory structure looks like

```
$ ls /opt/import_from
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
current
```

In order to import into Git, you need to review how Git stores its data. As you may remember, Git is fundamentally a linked list of commit objects that each point to a content snapshot. All you have to do is tell `git fast-import` what the content snapshots are, what commits points to them, and the order the commits go in. Your strategy will be to go through the snapshots one at a time and create commits with the contents of each directory, linking each commit back to the previous one.

As in the "An Example Git Enforced Policy" section of Chapter 7, I'll write this in Ruby, because it's what I generally work with. You can write this example pretty easily in any language you're familiar with. It just needs to send the appropriate information to stdout. And, if you're using Windows, this means you'll need to take special care to not introduce carriage returns (CR) at the end of lines. `git fast-import` is very particular about just wanting line feeds (LF), not the carriage return line feeds (CRLF) that Windows uses.

To begin, change into the target directory and identify every subdirectory, each of which is a snapshot that you want to import as a commit. Change into each subdirectory and output the commands necessary to export it. Your basic main loop looks like

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

Run `print_export` inside each directory, which takes the manifest and mark of the previous snapshot and returns the manifest and mark of the current one. That way, you can link the commits properly. A "mark" is the `git fast-import` term for an identifier you give to a commit. As you create commits, give each one a mark that you can use to link to it from other commits. So, the first thing to do in your `print_export` method is generate a mark from the directory name.

```
mark = convert_dir_to_mark(dir)
```

Do this by creating an array of directories and using the index value as the mark, because a mark must be an integer. Your method looks like

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Now that you have an integer representation of your commit, you need a date for the commit metadata. Because the date is expressed in the name of the directory, you can use that. The next line in your `print_export` file is

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

That returns an integer value for the date of each directory. The last piece of meta-information you need for each commit is the committer data, which you hardcode in a global variable:

```
$author = 'Scott Chacon <schacon@example.com>'
```

Now you're ready to begin outputting the commit data for your importer. The initial information states that you're defining a commit object and what branch it's on, followed by

the mark you've generated, the committer information and commit message, and then the previous commit, if any. The code looks like

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

Hardcode the time zone (-0700) because doing so is easy. If you're importing from a system in a different time zone than yours, you must specify the time zone as an offset. The commit message must be expressed in a special format like

```
data (size)\n(contents)
```

The format consists of the word `data`, the size of the data to be read, a newline character, and finally the data. Because you need to use the same format to specify the file contents later, create a helper method, `export_data`.

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

All that's left is to specify the file content for each snapshot. This is easy, because you have each snapshot in a directory. Print the `deleteall` command followed by the contents of each file in the directory. Git will then record each snapshot appropriately.

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Note: Because many VCSs think of revisions as changes from one commit to another, `git fast-import` accepts commands with each commit to specify which files have been added, removed, or modified, and what the new contents are. You could calculate the differences between snapshots and provide only this data, but doing so is more complex. You may as well give Git all the data and let it figure out what to do. If this approach is better suited to your data, check the `git fast-import` man page for details about how to provide your data in this manner.

The format for specifying new or modified file contents is as follows:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Here, 644 is the mode (if you detect an executable file, specify 755 instead), and inline says you'll list the contents immediately after this line. Your `inline_data` method looks like this:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Reuse the `export_data` method you defined earlier, because it uses the same technique for specifying commit message data.

The last thing to do is to return the current mark so it can be passed to the next iteration.

```
return mark
```

NOTE: If you're using Windows, add one extra step. As mentioned before, Windows uses CRLF to show the end of a line while `git fast-import` expects only LF. To get around this problem and make Git happy, tell Ruby to use LF instead of CRLF:

```
$stdout.binmode
```

That's it. If you run this script, you'll get content that looks something like

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
data 12
version two
```

```
commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
M 644 inline file.rb
data 14
version three
M 644 inline new.rb
data 16
new version one
(...)
```

To run the importer, pipe this output through `git fast-import` while in the Git repository you want to import into. Create a new directory and then run `git init` in it to start, and then run your script.

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:       5000
Total objects:           18 (         1 duplicates                  )
      blobs  :            7 (         1 duplicates          0 deltas)
      trees  :            6 (         0 duplicates          1 deltas)
      commits:            5 (         0 duplicates          0 deltas)
      tags   :            0 (         0 duplicates          0 deltas)
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              3
Memory total:          2255 KiB
       pools:          2098 KiB
     objects:           156 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize =   33554432
pack_report: core.packedGitLimit      =  268435456
pack_report: pack_used_ctr            =          9
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =       1356 /       1356
```

```
---------------------------------------------------------------------
```

As you can see, when your script completes successfully, it gives you a bunch of statistics about what it accomplished. In this case, you imported 18 objects total for 5 commits into 1 branch. Now, run `git log` to see your new history.

```
$ git log -2
commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700

    imported from current

commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700

    imported from back_2009_02_03
```

There you go — a nice, clean Git repository. It's important to note that nothing is checked out. You don't have any files in your working directory yet. To populate your working directory, reset your branch to where `master` is now.

```
$ ls
$ git reset --hard master
HEAD is now at 10bfe7d imported from current
$ ls
file.rb  lib
```

You can do a lot more with the `fast-import` tool — handle different modes, binary data, multiple branches and merging, tags, progress indicators, and more. A number of examples of more complex scenarios are available in the `contrib/fast-import` directory of the Git source code. One of the better ones is the `git-p4` script I just covered.

8.3 Summary

You should feel comfortable using Git with Subversion or importing nearly any existing repository format into Git without losing data. The next chapter will cover the raw internals of Git so you can craft every single byte, if need be.

Chapter 9

Git Internals

You may have skipped to this chapter from a previous chapter, or maybe you came here after reading the rest of the book. In either case, this is where I'll go over the inner workings and implementation details of Git. I found that learning this information is critical for understanding how to use Git, but others have argued that this information can be confusing and unnecessarily complex for beginners. Thus, I've made this discussion the last chapter in the book so you could read it at any point in your learning process. It's up to you to decide.

Now that you're here, let's get started. First, if it isn't yet clear, Git is fundamentally a content-addressable filesystem with a VCS on top of it. You'll learn more about what this means in a bit.

In the early days of Git (mostly pre-1.5), the user interface to the VCS was much more complex because it emphasized the filesystem rather than presenting a polished VCS. In the last few years, the UI has been refined until it's as clean and easy to use as any VCS out there. But, Git's reputation still suffers from its early complex and difficult to learn UI.

The content-addressable filesystem layer is amazingly cool, so I'll cover that first. Then, you'll learn about the transport protocols and the repository maintenance tasks that you may eventually have to deal with.

9.1 Plumbing and Porcelain

This book covers how to use 30 or so Git commands such as `git checkout`, `git branch`, `git remote`, and so on. But because Git was initially a toolkit for a VCS rather than a complete user-friendly VCS, it also contains a bunch of commands that do low-level work. These commands were designed to be chained together UNIX style or called from scripts, and are generally referred to as "plumbing" commands. The more user-friendly commands are called "porcelain" commands.

The book's first eight chapters deal almost exclusively with porcelain commands. This chapter, however, deals mostly with the lower-level plumbing commands because they illustrate the inner workings of Git and help demonstrate how and why Git does what it does. These commands aren't meant to be used directly on the command line, but rather to be used as building blocks for new tools and custom scripts.

When you run `git init`, Git creates a `.git` directory in the current directory. The `.git` directory is where almost everything that Git stores and manipulates is located. If you want

to back up or clone your repository, copying `.git` includes nearly everything you need. This entire chapter basically deals with the stuff in this directory.

A fresh `.git` directory contains

```
$ ls
HEAD
branches/
config
description
hooks/
index
info/
objects/
refs/
```

You may see some other files, but this is what you see by default. The `branches` directory isn't used by newer Git versions, and the `description` file is only used by the GitWeb program, so I won't cover them. The `config` file contains your project-specific configuration options, and the `info` directory keeps a global exclude file containing ignore patterns that you don't want in a .gitignore file. The `hooks` directory contains client- or server-side hook scripts, which were discussed in detail in Chapter 7.

This leaves the `HEAD` and `index` files, and the `objects` and `refs` directories. These are the core parts of Git. The `objects` directory stores the database content, the `refs` directory stores pointers to branches, the `HEAD` file points to the branch you're currently working on, and the `index` file is where Git stores staging area information. I'll now describe each of these sections in detail.

9.2 Git Objects

Git is a content-addressable filesystem. Great. What does that mean? It means that at the core of Git is a simple key-value data store. You can insert any kind of content into it, and Git will give you back a key that you can use to retrieve the content at any time. Git stores the content in the data store as objects, which I'll describe in great detail below.

The first kind of object you should be aware of is the `blob` object. This is the object type that Git uses to store files and directories. To demonstrate, use the plumbing command `git hash-object`, which takes some data, stores it in your `.git` directory as a blob object, and gives you back the key associated with the object. First, initialize a new Git repository and examine what's in the `objects` directory.

```
$ cd /tmp
$ mkdir test
$ cd test
$ git init
```

```
Initialized empty Git repository in /tmp/test/.git/
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Git has initialized the `objects` directory and created empty `pack` and `info` directories in it. Now, store some text in your Git database.

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

The `-w` option tells `git hash-object` to store the object. Otherwise, the command simply says what the key would be. The `--stdin` option tells the command to read the content from stdin. If you don't specify this, the command expects a filename. The output from the command is a 40-character SHA-1 hash. This is the checksum of the content you're storing, plus a header, which you'll learn about in a bit. This command inserted a blob object into your data store which you can see by running

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

You see a file in a subdirectory a couple of levels under the `objects` directory. This is how Git stores the blob object — as a single file per object whose name is the SHA-1 hash of the content and its header. The subdirectory name is the first 2 characters of the SHA-1 hash, and the filename is the remaining 38 characters. The content of the file is what you entered on stdin above.

Conversely, you can extract the object content with the `git cat-file` command. This command is sort of a Swiss army knife for inspecting Git objects. Passing `-p` to this command displays the content in a way that makes sense for the object type. For example, if the object is a blob containing a file, its content would be shown as follows:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Other object types, which you'll learn about below, are displayed differently.

By manually running Git plumbing commands you can do some simple version control. First, create a new file and save its contents in your Git database.

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Then, write some new contents to the file, and save it again.

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Your database now contains blob objects for all three versions of the file.

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Now, revert the file to the version containing `version 1`.

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

or the version containing `version 2`.

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Git can tell you the object type of any object, given its SHA-1 hash, when you run `git cat-file -t`.

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Notice that the blob object doesn't contain the filename of the file whose contents the object contains. Filenames are stored in tree objects, which I present next.

9.2.1 Tree Objects

A tree object contains one or more entries, each of which contains the SHA-1 hash of a blob or of another tree object, along with the mode, object type, and file or directory name. For example, a tree object in the simplegit project could contain the following:

```
$ git cat-file -p 7894138638f9404098a56befa8d6a4fc2598c087
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

You might notice the similarity between Git tree and blob objects, and a UNIX filesystem. The tree objects are like directories while the blob objects corresponding more or less to files.

Notice that entry for the `lib` directory isn't a blob but a pointer to another tree object. Examining its contents shows the contents of the `lib` directory.

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

The mode value in a tree object isn't the same as a Unix file protection mode value, although they look similar. There are only three modes valid for files (blobs) — `100644` for a normal file, `100755` for an executable file, and `120000` for a symbolic link. Other modes are used for directories (trees).

Conceptually, the data that Git is storing looks something like Figure 9-1.

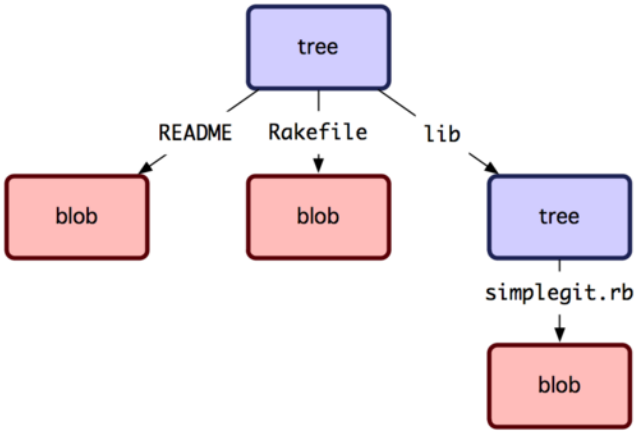

Figure 9.1: Simple version of the Git data model.

You can create your own tree object by using Git plumbing commands. Git normally writes a tree object by examining the state of your staging area and creating a tree object that reflects its structure. So, to create a tree object, first set up a staging area by adding some files to it. To create a staging area with a single entry — the first version of your `test.txt` file — use the plumbing command `git update-index`. Use this command to artificially add the earlier version of the `test.txt` file to an empty staging area. You must pass the `--add`

option because the file doesn't yet exist in your staging area, and `--cacheinfo` because the file you're adding isn't in your directory but is in your database. Then, specify the mode, SHA-1 hash, and filename.

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Now, use the `git write-tree` command to write the staging area out to a tree object. No `-w` option is needed — running `git write-tree` automatically creates a tree object from the state of the staging area if that tree doesn't yet exist.

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

You can also verify that this is a tree object.

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Now create a new tree object with the second version of test.txt and a new file as well.

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Your staging area now has the new version of `test.txt` as well as the new file `new.txt`. Create a tree object that reflects the current state of the staging area and see what it looks like.

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Notice that this tree contains both file entries and also that the `test.txt` SHA-1 hash is the "version 2" SHA-1 hash from earlier (`1f7a7a`). Just for fun, add the first tree as a

subdirectory into this one. Read trees into your staging area by running `git read-tree`. In this case, read an existing tree into your staging area as a subtree by using the `--prefix` option to `git read-tree`.

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

If you created a working directory from the new tree you just wrote, you would get the two files in the top level of the working directory and a subdirectory named `bak` that contained the first version of the test.txt file. You can think of the objects that Git contains now as being like Figure 9-2.

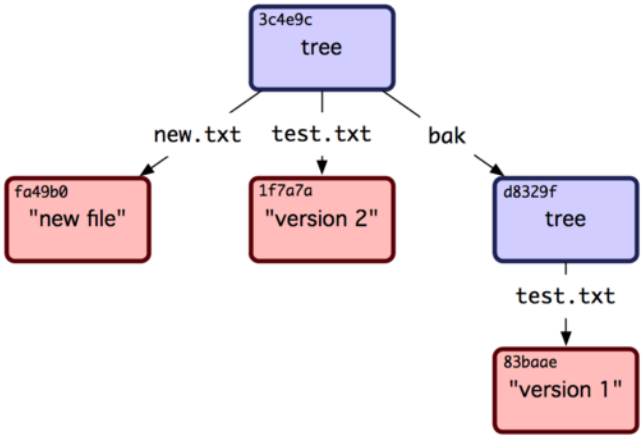

Figure 9.2: The content structure of your current Git data.

9.2.2 Commit Objects

You created three trees that contain the SHA-1 hashes of the blob objects representing the files you want to track. Each tree is said to point to a `snapshot` of the contents of the index at the time you created the tree. But the earlier problem remains — you must remember all three SHA-1 hashes in order to retrieve the snapshots. You also don't have any information about who saved the snapshots, when they were saved, or why they were saved. This is the basic information that the commit object stores.

To create a commit object, run `git commit-tree` and specify the SHA-1 hash of a single tree and the commit objects, if any, that directly precede it. The first commit in a repository won't have any preceding commits. Start with the first tree you wrote.

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Now look at your new commit object by running `git cat-file`.

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

The format of a commit object is simple. It specifies the SHA-1 hash for the top-level tree of the project at that point — the author/committer information pulled from your `user.name` and `user.email` configuration settings, combined with the current timestamp, a blank line, and then the commit message.

Next, create the other two commit objects, each referencing the commit that came directly before it.

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Each of the three commit objects points to one of the three snapshot trees objects you created. Oddly enough, you have a real Git history now that you can view with the `git log` command, if you pass it the last commit object's SHA-1 hash.

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    third commit

 bak/test.txt |    1 +
 1 file changed, 1 insertion(+), 0 deletions(-)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
```

```
Date:   Fri May 22 18:14:29 2009 -0700

    second commit

 new.txt  |    1 +
 test.txt |    2 +-
 2 files changed, 2 insertions(+), 1 deletions(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

 test.txt |    1 +
 1 file changed, 1 insertion(+), 0 deletions(-)
```

Amazing. You've just performed the low-level operations to build a Git commit history without using any of the Git porcelain commands. This is essentially what Git does when you run the `git add` and `git commit` commands. Git stores blobs representing the files that have changed, updates the staging area, writes out trees, and writes commit objects that reference the top-level trees and the commits that came immediately before them. These three main Git objects — the blob, the tree, and the commit — are initially stored as separate files in your `.git/objects` directory. Here are all the objects in the example `.git/objects` directory now, with comments saying what they store:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

If you follow all the internal pointers, you get an object graph looking something like Figure 9-3.

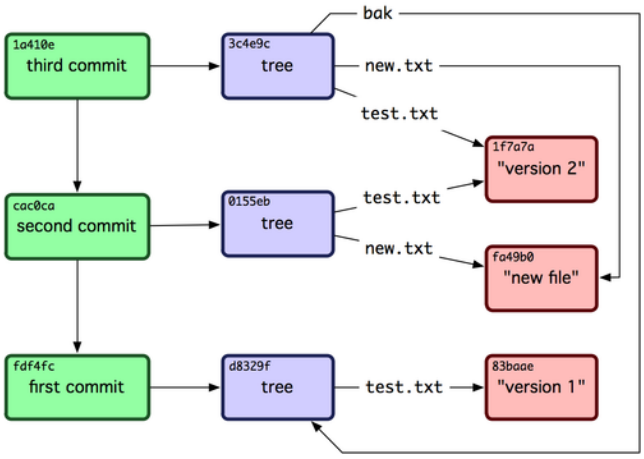

Figure 9.3: All the objects in your Git directory.

9.2.3 Object Storage

I mentioned earlier that a header is stored with the content when creating a blob object. Let's take a minute to look at how Git stores its objects. You'll see how to store a blob object — in this case, the string "what is up, doc?" — interactively in the Ruby scripting language. Start up interactive Ruby mode with the `irb` command.

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git constructs a header that starts with the type of the object, in this case `blob`. Then, it adds a space followed by the size of the content, and finally a null byte.

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git concatenates the header and the original content into what I'll call the new content. It then calculates the SHA-1 hash of the new content. You can calculate the SHA-1 hash of a string in Ruby by including the SHA1 digest library with the `require` statement and then calling `Digest::SHA1.hexdigest()`, passing the string to hash.

```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git compresses the new content with zlib, which you can also do in Ruby. First, require the zlib library and then run `Zlib::Deflate.deflate()` on the new content.

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\234K\312\311OR04c(\317H,Q\310,V(-\320QH\311O\266\a\000_\034\a\235"
```

Finally, write your zlib-compressed new content to a Git object. Determine the path of the object you want to write out (the first two characters of the SHA-1 hash as the subdirectory name, and the last 38 characters as the filename in that subdirectory). In Ruby, use the `FileUtils.mkdir_p()` function to create the subdirectory if it doesn't exist. Then, open the file with `File.open()` and write out the previously zlib-compressed new content to the file with a `write()` call on the resulting file handle.

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

That's it! You've created a valid Git blob object. All Git objects are stored the same way, just with different types. Instead of the string `blob`, the header will begin with `commit` or `tree`. Also, although the blob object content can be nearly anything, the commit and tree object content have specific requirements.

9.3 Git References

You can run something like `git log 1a410e` to look through your whole history, but you still have to remember that `1a410e` is the last commit in order to know where to start to see all those objects. Git also allows creating a file with a specific name whose content helps you remember a specific commit. The SHA-1 hash of the commit is the content of the file.

In Git, these are called "references" or "refs". The files that contain the SHA-1 hashes are in the `.git/refs` directory. In the current project, this directory contains no files, but it has a simple structure.

```
$ find .git/refs
.git/refs
```

```
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

To create a new reference to help remember where your latest commit is, you can technically do something as simple as

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

This created a head reference, because the SHA-1 hash you put in the file points to the head of a series of commit objects. The name of the reference is the final component in the filename — in this case `master`.

Now, use `master` instead of the SHA-1 hash in your Git commands.

```
$ git log --pretty=oneline  master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You aren't encouraged to directly edit reference files. Instead Git provides a safer command called `git update-ref` to update a reference.

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

That's basically what a branch in Git is — a simple pointer or reference to the head of a list of commits. To create a branch back at the second commit, run

```
$ git update-ref refs/heads/test cac0ca
```

Your branch will contain only work starting from that commit.

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, your Git database conceptually looks something like Figure 9-4.

When you run commands like `git branch (branchname)`, Git basically runs `git update-ref` to add the SHA-1 hash of the last commit of the branch you're on into whatever new reference you want to create.

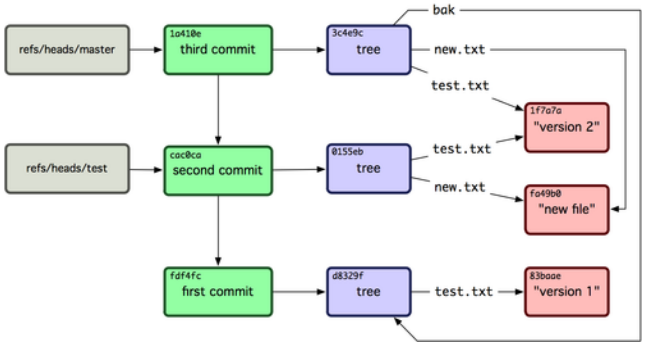

Figure 9.4: Git directory objects with branch head references included.

9.3.1 The HEAD

The question now is, when you run `git branch (branchname)`, how does Git know the SHA-1 hash of the last commit? The answer is the HEAD file, which is a symbolic reference to the branch you're currently on. By symbolic reference, I mean that unlike a normal reference, it doesn't contain a SHA-1 hash but rather the name of another reference. If you look in this file, you'll normally see something like

```
$ cat .git/HEAD
ref: refs/heads/master
```

If you run `git checkout test`, Git updates `.git/HEAD` to look like

```
$ cat .git/HEAD
ref: refs/heads/test
```

When you run `git commit`, Git creates the commit object, specifying the SHA-1 hash of the reference HEAD points to as the parent commit object.

You can also manually edit this file, but `git symbolic-ref` is a safer command for doing this. You can see the value of HEAD using the following command:

```
$ git symbolic-ref HEAD
refs/heads/master
```

You can also set the value of HEAD.

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

You can't set a symbolic reference outside of the refs style.

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

9.3.2 Tags

I've just gone over Git's three main object types, but there's also a fourth type — the tag object. The tag object is very much like a commit object. It contains the username of the person doing the tagging, a date, a message, and a pointer to the object being tagged. The main difference is that a tag object points to a commit rather than a tree. It's like a branch reference, but it never moves. It always points to the same commit but lets you use a friendlier name to refer to the commit.

As discussed in Chapter 2, there are two types of tags: annotated and lightweight. You make a lightweight tag by running something like

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

That's all a lightweight tag is — a branch that never moves. An annotated tag is more complex, however. If you create an annotated tag, Git creates a tag object and then creates a reference to point to the tag object rather than directly to the commit. You can see this by creating an annotated tag (`-a` specifies an annotated tag).

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Here's the object SHA-1 hash it created.

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Now, run `git cat-file` on that SHA-1 hash.

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

Notice that the object entry points to the commit SHA-1 hash that you tagged. Also notice that it doesn't need to point to a commit. You can tag any Git object. In the Git source code, for example, the maintainer has added his GPG public key as a blob object and then tagged it. You can view the public key by running

```
$ git cat-file blob junio-gpg-pub
```

in the Git source code repository. The Linux kernel repository also has a non-commit-pointing tag object. The first tag created points to the initial tree of the import of the source code.

9.3.3 Remotes

The third type of reference is a remote reference. If you add a remote and push to it, Git stores in the `refs/remotes` directory the SHA-1 hash of the most recent commit from each branch you pushed to that remote. For instance, add a remote called `origin` and push your `master` branch to it.

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
   a11bef0..ca82a6d  master -> master
```

The value in the `refs/remotes/origin/master` file then contains the SHA-1 hash of the `master` branch on the remote `origin` server.

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remote references differ from branches (`refs/heads` references) mainly in that remote references can't be checked out. Git moves them around as bookmarks to the last known state of where those branches were on remote servers.

9.4 Packfiles

Let's go back to the objects database for your test Git repository. At this point, you have 11 objects — 4 blobs, 3 trees, 3 commits, and 1 tag.

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git compresses the contents of these files with zlib, and these files aren't very big anyway, so they collectively occupy only 925 bytes. Add some larger content to the repository to demonstrate an interesting feature of Git. Add the `repo.rb` file from the Grit library you worked with earlier — about a 12K file.

```
$ curl https://raw.github.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 459 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

If you look at the resulting tree, you see the SHA-1 hash of the blob object holding the contents of `repo.rb`.

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Then check how big that object is on disk.

```
$ du -b .git/objects/9b/c1dc421dcd51b4ac296e3e5b6e2a99cf44391e
4102 .git/objects/9b/c1dc421dcd51b4ac296e3e5b6e2a99cf44391e
```

Now, modify that file a little, and see what happens.

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master ab1afef] modified repo a bit
 1 file changed, 1 insertion(+), 0 deletions(-)
```

Checking the tree created by that commit shows something interesting.

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

The blob containing `repo.rb` is now different, which means that although you added only a single line to the end of a 400-line file, Git stored that new content in a completely new object.

```
$ du -b .git/objects/05/408d195263d853f09dca71d55116663690c27c
4109 .git/objects/05/408d195263d853f09dca71d55116663690c27c
```

You now have two nearly identical 4K objects on your disk. Wouldn't it be nice if Git could store one of them in full but then store the second object as only the difference between it and the first?

It turns out that this is exactly what Git does. The initial format in which Git saves objects is called loose object format. However, occasionally Git packs several loose objects into a single file called a packfile in order to save space and be more efficient. Git does this if you have too many loose objects around, if you push to a remote server, or if you run the `git gc` command manually. Let's see what happens when you run `git gc`.

```
$ git gc
Counting objects: 17, done.
Delta compression using 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

If you look in your objects directory, you'll find that most of your objects are gone, and a new pair has appeared.

```
$ find .git/objects -type f
.git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

The objects that remain are the blobs that aren't pointed to by any commit — in this case, the "what is up, doc?" and the "test content" blobs you created earlier. Because you never added them to any commits, they're considered dangling and aren't included in your new packfile.

The two new files are your new packfile and an index. The packfile is a single file containing all the objects that were removed from your objects directory. The index contains offsets into that packfile so Git can quickly seek to a specific object. What's cool is that although the objects on disk before you ran `git gc` were collectively about 8K in size, the new packfile is only 4K. You've halved your disk usage by packing your objects.

How does Git do this? When Git packs objects, it looks for files that are named and sized similarly, and stores just the differences from one version of the file to the next. You can examine the packfile to see what Git did to save space. The `git verify-pack` plumbing command allows you to see what was packed.

```
$ git verify-pack -v \
  .git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 5400
05408d195263d853f09dca71d55116663690c27c blob   12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree   106 107 5086
1a410efbd13591db07496601ebc7a059dd55cfe9 commit 225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   101 105 5211
484a59275031909e19aadb7c92262719cfcdf19a commit 226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag    136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob   7 18 5193 1 \
  05408d195263d853f09dca71d55116663690c27c
ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit 226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob   1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree   106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok
```

Here, the `9bc1d` blob, which, if you remember, was the first version of `repo.rb`, is referencing the `05408` blob, which was the second version of the file. The third column in the output is the size of the object's content, so you can see that the content of `05408` takes up 12K, but that of `9bc1d` only takes up 7 bytes. What's also interesting is that the second version of the file is the one that's stored intact, whereas the original version is stored as a delta. This is because you're most likely to need faster access to the most recent version.

The really nice thing about this technique is that Git can automatically repack objects at any time, always trying to save more space. You can also repack at any time by running `git gc` by hand.

9.5 The Refspec

Throughout this book, you've used simple mappings from remote branches to local references but they can be more complex. Suppose you add a remote like

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

This adds a section to your `.git/config` file, specifying the name of the remote (`origin`), the URL of the remote repository, and the refspec for fetching.

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/*:refs/remotes/origin/*
```

The format of the refspec is an optional `+`, followed by `<src>:<dst>`, where `<src>` is the pattern for references on the remote side and `<dst>` is where those references will be written locally. The `+` tells Git to update the reference even if it isn't a fast-forward.

In the default case that's automatically occurs when you run `git remote add`, Git fetches all the references under `refs/heads/` on the remote server and writes them to `refs/remotes/origin/` locally. So, if there's a `master` branch on the remote server, you can access the log of that branch locally via

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

They're all equivalent, because Git expands each of them to `refs/remotes/origin/master`.

If you want Git instead to pull only the `master` branch, rather than every branch on the remote server, change the fetch line to

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

This is just the default refspec for `git fetch` for that remote. To fetch a branch only once, specify the full refspec on the command line. For example, to pull the `master` branch on the remote server to `origin/mymaster` locally, run

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

You can also specify multiple refspecs. On the command line, pull down several branches like so.

```
$ git fetch origin master:refs/remotes/origin/mymaster \
   topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]        master     -> origin/mymaster  (non fast forward)
 * [new branch]      topic      -> origin/topic
```

In this case, the pull of the `master` branch was rejected because it wasn't a fast-forward reference. You can override that by including the `+` in front of the refspec.

You can also specify multiple refspecs for fetching your configuration file. To always fetch the `master` and `experiment` branches, add two lines.

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/master:refs/remotes/origin/master
       fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

You can't use partial globs in the pattern, so the following would be invalid:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

However, you can use namespacing to accomplish something similar. If you have a QA team that pushes a series of branches, and you want to pull the `master` branch and any of the QA team's branches but nothing else, create a config section containing

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/master:refs/remotes/origin/master
       fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

If you have a complex workflow that includes a QA team pushing branches, developers pushing branches, and integration teams pushing and collaborating on remote branches, namespace them easily this way.

9.5.1 Pushing Refspecs

It's nice that you can fetch namespaced references that way, but how does the QA team get their branches into a `qa/` namespace in the first place? They accomplish that by using refspecs when they push.

If the QA team wants to push their `master` branch to `qa/master` on the remote server, they run

```
$ git push origin master:refs/heads/qa/master
```

If they want Git to do that automatically each time they run `git push origin`, they add a `push` value to their config file.

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/*:refs/remotes/origin/*
       push = refs/heads/master:refs/heads/qa/master
```

Again, this will cause `git push origin` to push the local `master` branch to the remote `qa/master` branch by default.

9.5.2 Deleting References

You can also use the refspec to delete references from the remote server by running something like

```
$ git push origin :topic
```

Because the refspec is `<src>:<dst>`, leaving off the `<src>` part says to make the topic branch on the remote nothing, which deletes it.

9.6 Transfer Protocols

Git can transfer data between repositories in two primary ways: over HTTP and via the so-called smart protocols used by the `file://`, `ssh://`, and `git://` transports. This section quickly covers how these two styles operate.

9.6.1 The Dumb Protocol

Git transport over HTTP is often referred to as a dumb protocol because it doesn't run any Git code on the remote server during the transport process. Fetching is simply a series of HTTP GET requests, where the client can replicate the layout of the Git repository that appears on the server. Let's follow the `http-fetch` process for the simplegit library.

```
$ git clone http://github.com/schacon/simplegit-progit.git
```

The first thing this command does is pull down the `info/refs` file. This file is updated by the `update-server-info` command, which for the HTTP transport to work properly needs to be enabled as a `post-receive` hook:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

Now you have a list of the remote references and SHA-1 hashes. Next, Git looks for what the HEAD reference points to so it knows what to check out when the transfer is finished.

```
=> GET HEAD
ref: refs/heads/master
```

At this point, Git is ready to start the repository walking process. Because the starting point is the `ca82a6` commit object found in the `info/refs` file, Git starts by fetching that object.

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

The object that's returned is in loose format on the server, and Git fetched it using a static HTTP GET request. Git can zlib-uncompress it, strip off the header, and look at the commit content.

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Next, Git has two more objects to retrieve — `cfda3b`, which is the tree object that the commit just retrieved points to, and `085bb3`, which is the parent commit.

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

That gives the next commit object. Git grabs the tree object.

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oops — it looks like that tree object isn't in loose format on the server, so a 404 response is returned. There are a couple of reasons for this. The object could be in an alternate repository, or it could be in a packfile in this repository. Git checks for any listed alternates first.

```
=> GET objects/info/http-alternates
(empty file)
```

If this comes back with a list of alternate URLs, Git checks for loose files and packfiles in each one. This is a nice mechanism for projects that are forks of one another to share objects. However, because in this case no alternates are listed, your object must be in a packfile. To see what packfiles are available on this server, Git needs to retrieve the `objects/info/packs` file, which contains a listing of all the packfiles (also generated by `update-server-info`).

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

There's only one packfile on the server, so the object is obviously there, but Git checks the index file to make sure. This is also useful if there are multiple packfiles on the server, so Git can see which packfile contains the object it needs.

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Now that Git has read the packfile index, Git can search the index for the SHA-1 it's looking for. Remember, the index contains hashes of the objects contained in the packfile and the offsets to those objects. The object is there, so Git can go ahead and get the whole packfile.

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

Git found the tree object, and continues walking the commits. They're all also within the packfile just downloaded, so Git doesn't have to make any more requests to the server. Git checks out a working copy of the `master` branch that was pointed to by the HEAD reference it downloaded at the beginning.

The entire output of this process looks like

```
$ git clone http://github.com/schacon/simplegit-progit.git
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
got ca82a6dff817ec66f44342007202690a93763949
walk ca82a6dff817ec66f44342007202690a93763949
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Getting alternates list for http://github.com/schacon/simplegit-progit.git
Getting pack list for http://github.com/schacon/simplegit-progit.git
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
 which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

9.6.2 The Smart Protocol

The HTTP method is simple but inefficient. It's more common to use smart protocols to transfer Git data. These protocols require a Git process to be running on the remote end. The process can read server Git data and figure out what the client has or needs so that it can return just what's needed. There are actually two sets of processes for transferring data — a pair for uploading data and a pair for downloading data.

Uploading Data

To upload data to a remote process, Git uses the `send-pack` and `receive-pack` processes. The `send-pack` process runs on the client and connects to a `receive-pack` process on the remote end.

For example, say you run `git push origin master`, and `origin` is defined as a URL that uses the SSH protocol. Git fires up the `send-pack` process, which initiates a connection over SSH to the remote server. `send-pack` tries to run a command on the remote server via an SSH call that looks something like

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949       refs/heads/master       report-
status delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

The `git-receive-pack` command immediately responds with one line for each reference currently in the remote repository — in this case, just the `master` branch and its SHA-1 hash. The first line also includes a list of the server's capabilities (here, `report-status` and `delete-refs`).

Each line starts with a 4-byte hex value specifying the length of the rest of the line. The first line starts with 005b in hex, which is 91 in decimal, meaning that 91 bytes remain on that line. The next line starts with 003e, which is 62, so Git reads the remaining 62 bytes. The next line is 0000, meaning the server is done listing references.

Now that the `send-pack` process knows the server's state, it determines which commits the client has that the server doesn't. The `send-pack` process tells the `receive-pack` process about each remote reference that this push will update. For instance, if you're updating the `master` branch and adding an `experiment` branch, the `send-pack` response may look something like

```
0085ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6 refs/
heads/master report-status
00670000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d refs/
heads/experiment
0000
```

The SHA-1 hash of all '0's means that nothing was there before, because you're adding the `experiment` reference. If you were deleting a reference, you would see the opposite — all '0's on the right side.

For each reference you're updating Git sends a line containing the old SHA-1 hash, the new SHA-1 hash, and the reference that's being updated. The first line also shows the client's capabilities. Next, the client uploads a packfile containing all the objects the server doesn't have yet. Finally, the server responds with a success (or failure) indication.

```
000Aunpack ok
```

Downloading Data

When Git downloads data, the `fetch-pack` and `upload-pack` processes are run. The client initiates a `fetch-pack` process that connects to an `upload-pack` process on the remote server in order to negotiate what data will be transferred.

There are different ways to initiate the `upload-pack` process on the remote server. It can start via SSH the same way as the `receive-pack` process. Or, if the remote server is running the Git daemon, you can initiate the transfer process by connecting to port 9418. The `fetch-pack` process sends data that looks like this to the daemon after connecting.

```
003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

It starts with 4 bytes specifying how much data follows, then the command to run followed by a null byte, and then the server's hostname followed by a final null byte. The Git daemon checks that the command can be run, that the repository exists, and that the repository is publicly accessible. If everything passes, the Git daemon fires up the `upload-pack` process and hands off the request to it.

If you're doing the fetch over SSH, `fetch-pack` instead runs something like

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

In either case, after `fetch-pack` connects, `upload-pack` sends back something like

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
  side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

This is very similar to how `receive-pack` responds, but the capabilities are different. In addition, `upload-pack` sends back the HEAD reference so the client knows what to check out if this is a clone.

At this point, the `fetch-pack` process looks at the objects it has and responds with the objects that it needs by sending "want" and then the SHA-1 hash of the object it wants. It then sends all the objects it already has with "have" and then the SHA-1 hashes of the objects. At the end of this list, it sends "done" to tell the `upload-pack` process to begin sending the packfile of the data it needs.

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

These are very basic examples of the transfer protocols. In more complex cases, the client supports `multi_ack` or `side-band` capabilities but these examples show you the basic back and forth used by the smart protocol processes.

9.7 Maintenance and Data Recovery

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

9.7.1 Maintenance

Occasionally, Git runs `git gc --auto`. Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged garbage collection pass, which does a number of things. It gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren't reachable from any commit and are a few months old.

You can run auto gc manually.

```
$ git gc --auto
```

Again, this does nothing unless you have around 7,000 loose objects or more than 50 packfiles. You can modify these limits with the `gc.auto` and `gc.autopacklimit` config settings, respectively.

The other thing `git gc` does is pack up your references into a single file. Suppose your repository contains the following branches and tags:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

If you run `git gc`, you'll no longer have these files in the `refs` directory. For the sake of efficiency Git will move them into a file named `.git/packed-refs` that looks like

```
$ cat .git/packed-refs
# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

If you update a reference, Git doesn't edit `.git/packed-refs` but instead writes a new file in `refs/heads`. To get the appropriate SHA-1 hash for a given reference, Git checks for that reference in the `refs` directory and then checks the `packed-refs` file as a fallback. However, if Git can't find a reference in the `refs` directory, it's probably in the `packed-refs` file.

Notice the last line of the file, which begins with a `^`. This means the tag directly above is an annotated tag and that line is the commit that the annotated tag points to.

9.7.2 Data Recovery

At some point in your Git journey, you may accidentally lose a commit. Generally, this happens because you force-delete a branch that had work on it, and it turns out you wanted the branch after all. Or, you hard-reset a branch, thus abandoning commits that you still need. If this happens, how can you get your commits back?

Here's an example that hard-resets the master branch in your test repository to an older commit and then recovers the lost commits. First, let's review where your repository is at this point.

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, move the `master` branch back to the middle commit.

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You've effectively lost the top two commits. You have no branch from which those commits are reachable. You need to find the latest commit SHA-1 hash and then add a branch that points to it. The trick is finding that latest commit SHA-1 hash. You probably haven't memorized it, right?

The quickest way is often using a tool called `git reflog`. As you're working, Git silently records the value of HEAD every time it changes. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the `git update-ref` command, which is another reason to use it instead of just writing the SHA-1 hash value to your ref files, as I mentioned earlier in the "Git References" section of this chapter. You can see where you've been at any time by running `git reflog`.

```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

Here you can see the two commits that you checked out but there isn't much other information here. To see the same information in a much more useful way, run `git log -g`, which will give you a normal log output for your reflog.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

    third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

     modified repo a bit
```

It looks like the bottom commit is the one you lost, so you can recover it by creating a new branch at that commit. For example, start a branch named `recover-branch` at that commit (ab1afef).

```
$ git branch recover-branch ab1afef
```

```
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Cool. Now you have a branch named `recover-branch` that's where your `master` branch used to be, making the first two commits reachable again. Next, suppose, for some reason, the commits you lost aren't in the reflog — you can simulate that by removing `recover-branch` and deleting the reflog. Now the first two commits aren't reachable by anything.

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Because the reflog data is kept in the `.git/logs/` directory, you've just lost the reflog. How can you recover a commit at this point? One way is to use the `git fsck` utility, which checks the integrity of your repository. If you run `git fsck` with the `--full` option, you'll see all the objects that aren't pointed to by another object.

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In this case, you can see your missing commit after the dangling commit. You can recover it the same way, by adding a branch that points to that SHA-1 hash.

9.7.3 Removing Objects

There are a lot of great things about Git, but one feature that can cause issues is the fact that `git clone` downloads the entire history of the project, including every version of every file. This is fine if the whole thing is source code, because Git is highly optimized to compress source code efficiently. However, if someone at any point in the history of your project added a single huge file, every clone for all time will be forced to include that file, even if it was removed from the project in the very next commit. Because the commit that added the huge file is in the commit history, the file will always be there.

This can be a huge problem when you're importing Subversion or Perforce repositories into Git. If you find that your repository is much larger than it should be, here's how you can find and remove large objects.

Be warned: this technique is destructive to your commit history. It rewrites every commit object starting from the earliest tree containing a large file reference. If you do this immediately after an import, before anyone has started to base work on the commit, you're fine. Otherwise, you have to notify all contributors that they must rebase their work onto your new commits.

To demonstrate, add a large file into a test repository, remove it in the next commit, find it, and remove it permanently from the repository. First, add a large file.

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
$ git add git.tbz2
$ git commit -am 'added git tarball'
[master 6df7640] added git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tbz2
```

Oops — you didn't want to add a huge tarball to your project. Better get rid of it.

```
$ git rm git.tbz2
rm 'git.tbz2'
$ git commit -m 'oops - removed large tarball'
[master da3f30d] oops - removed large tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tbz2
```

Now, run `git gc` and see how much space you're using.

```
$ git gc
Counting objects: 21, done.
Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), done.
Total 21 (delta 3), reused 15 (delta 1)
```

Run `git count-objects` to quickly see how much space you're using:

```
$ git count-objects -v
count: 4
size: 16
in-pack: 21
packs: 1
```

```
size-pack: 2016
prune-packable: 0
garbage: 0
```

The `size-pack` entry is the size of your packfiles in kilobytes, so you're using 2MB. Before the last commit, you were using closer to 2K. Clearly, removing the file from the previous commit didn't remove it from your history. Every time anyone clones this repository, they'll have to clone all 2MB just to get this tiny project, because you accidentally added a big file. Let's get rid of it.

First you have to find it. In this case, you already know what file it is. But suppose you didn't know this. How would you identify what was taking up so much space? If you run `git gc`, all the objects are in a packfile. You can identify the big objects by running another plumbing command called `git verify-pack` and sorting on the third field of the output, which is file size. You can also pipe the output through the `tail` command because you're only interested in the last few largest files.

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -
n | tail -3
e3f094f522629ae358806b17daf78246c27c007b blob   1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob   12908 3478 1189
7a9eb2fba2b1811321254ac360970fc169ba2330 blob   2056716 2056872 5401
```

The big object is the 2MB object at the bottom. To find out what file it is, use the `git rev-list` command, which you used briefly in Chapter 7. If you pass `--objects` to `git rev-list`, it lists all the commit SHA-1 hashes and also the blob SHA-1 hashes with the file paths associated with them. Use this to find your blob's name.

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

Now, you need to remove this file from all trees in your history. You can easily see what commits added or remoted this file.

```
$ git log --pretty=oneline --branches -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

You must rewrite all the commits downstream from `6df76` to fully remove this file from your Git history. To do so, use `git filter-branch`, which you used in Chapter 6.

```
$ git filter-branch --index-filter \
   'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
Ref 'refs/heads/master' was rewritten
```

The `--index-filter` option is similar to the `--tree-filter` option used in Chapter 6, except that instead of passing a command that modifies checked out files, you're modifying your staging area each time. Rather than remove a specific file with something like `git rm file`, you have to remove it with `git rm --cached`. You must remove it from the index, not from your working directory. The reason for this is speed. Because Git doesn't have to check out each revision before running your filter, the process can be much, much faster. You can accomplish the same task with the `--tree-filter` option. The `--ignore-unmatch` option to `git rm` tells it not to abort if the pattern you're trying to remove isn't found. Finally, tell `git filter-branch` to rewrite your history only starting from the `6df7640` commit, because you know that's where this problem started. Otherwise, it will start from the first commit and will unnecessarily take longer.

Your history no longer contains a reference to that file. However, your reflog and the new set of refs that Git added when you ran `git filter-branch` under `.git/refs/original` still contain the references, so you have to remove them and then repack the database. You need to get rid of anything that has a pointer to those old commits before you repack.

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

Let's see how much space you saved.

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

The packed repository size is down to 7K, which is much better than 2MB. You can see from the size that the big object is still in your loose objects, so it's not gone. But it won't be transferred on a push or subsequent clone, which is what's important. If you really want, you could remove the file completely by running `git prune --expire`.

9.8 Summary

You should have a pretty good understanding of what Git does in the background and, to some degree, how it's implemented. This chapter has covered a number of plumbing commands — commands that are lower level and simpler than the porcelain commands you've learned about in the rest of the book. Understanding how Git works at this level should make it easier to understand why it's doing what it's doing. You should also be able to write your own tools and custom scripts to make your specific workflow work for you.

Git as a content-addressable filesystem is a very powerful tool that you can easily use as more than just a VCS. I hope you use your new found knowledge of Git internals to implement your own cool application of this technology and feel more comfortable doing so.