Pro Git Reedited

Jon Forrest

Pro Git Reedited

Jon Forrest

This book is for sale at http://leanpub.com/progitreedited

This version was published on 2013-12-15

Contents

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[1]

and the PDF version is at

https://www.dropbox.com/s/4awq55350ef235m/progitreedited.en.pdf[2]

[1]http://

[2]http://

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.

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.

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).

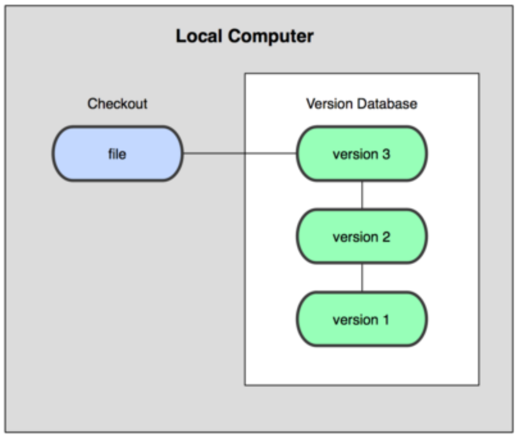


Figure 1-1. Local version control diagram.

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.

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).

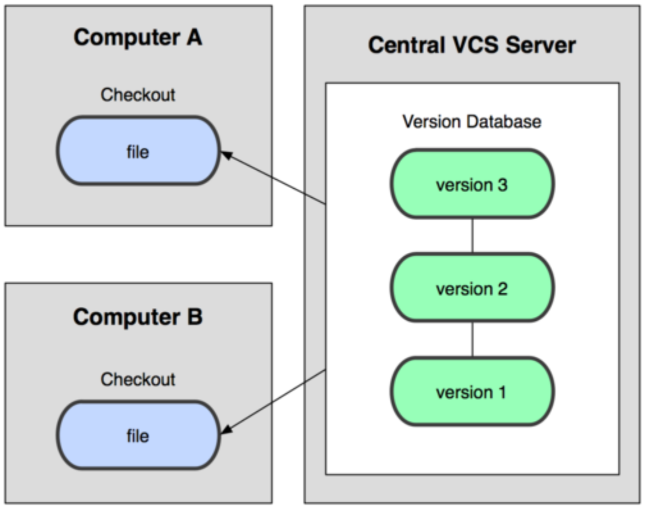


Figure 1-2. Centralized version control diagram.

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.

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).

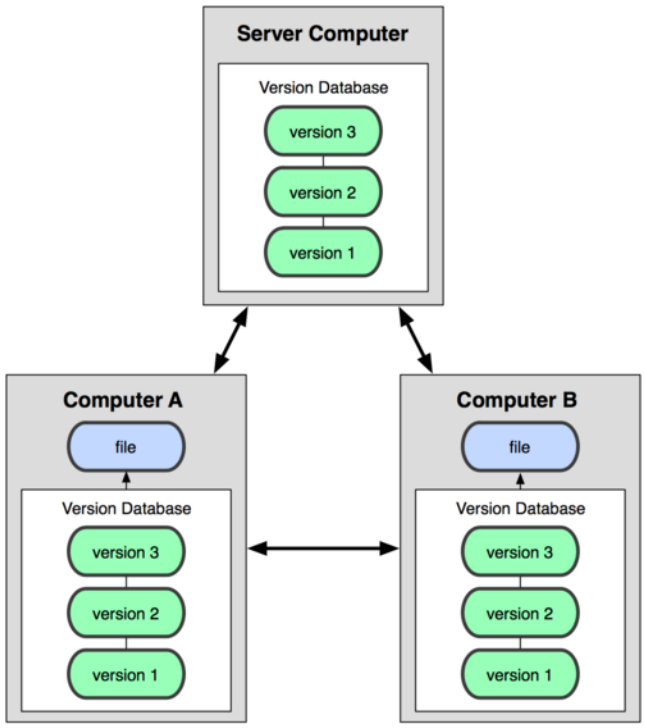


Figure 1-3. Distributed version control diagram.

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.

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*).

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.

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.

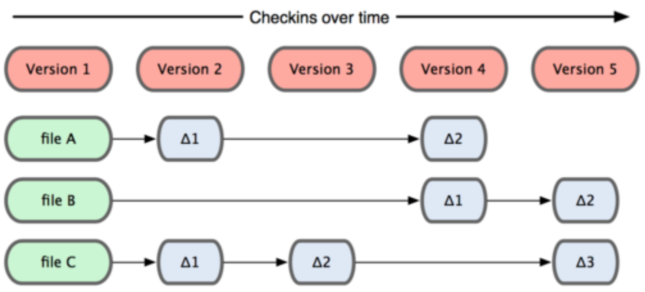


Figure 1-4. Other systems tend to store data as changes to a base version of each file.

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.

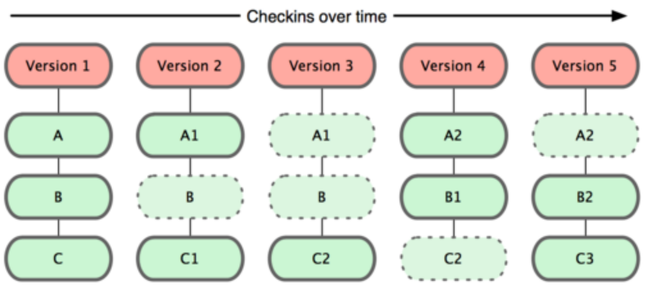


Figure 1-5. Git stores data as snapshots of the project over time.

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*.

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*.

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.

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.

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*.

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.

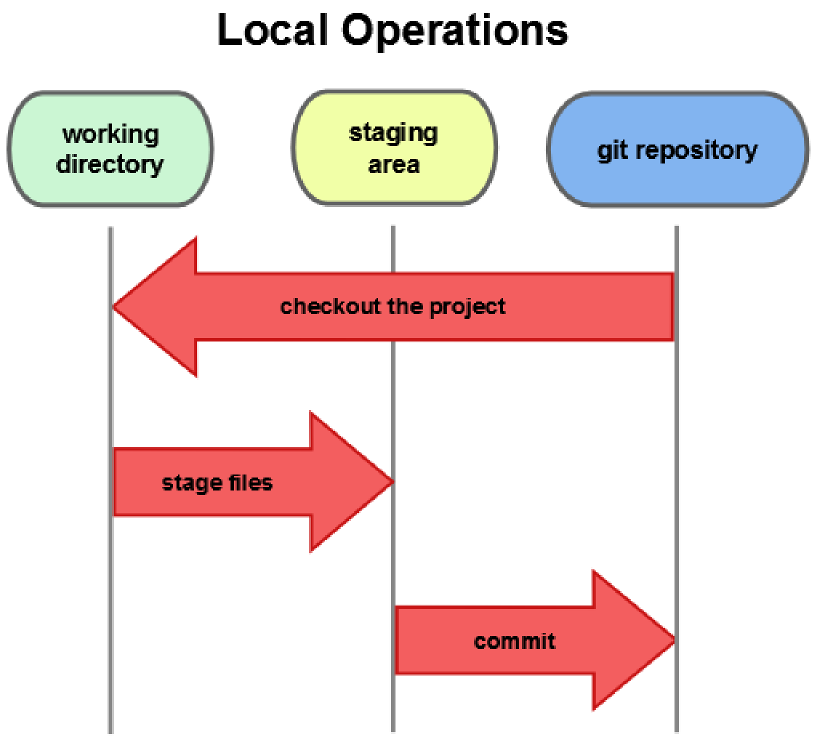


Figure 1-6. Working directory, staging area, and Git repository.

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.

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.

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*.

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.

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
```

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
```

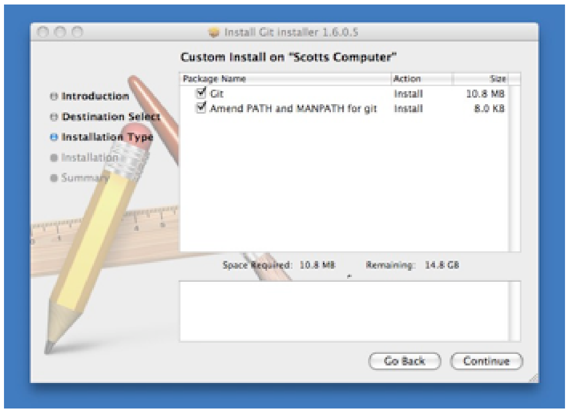
Or, if you're on a Debian-based distribution like Ubuntu, try apt-get.

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

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
```



Custom Install on "Scotts Computer"

Package Name	Action	Size
✓ Git	Install	10.8 MB
✓ Amend PATH and MANPATH for git	Install	8.0 KB

Space Required: 10.8 MB Remaining: 14.8 GB

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*).

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.

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.

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.

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
```

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).

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
```

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.

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.

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.

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.

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.

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.

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.

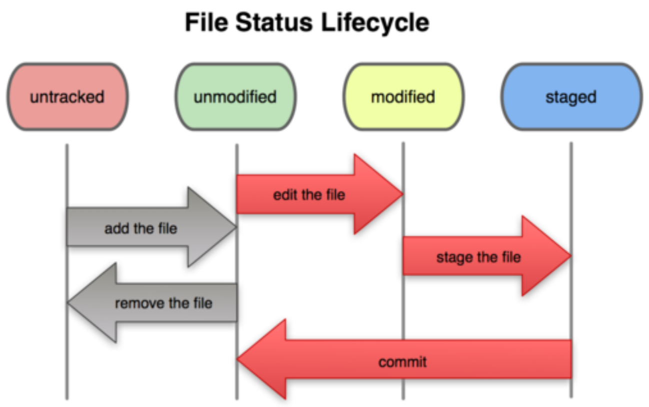


Figure 2-1. The lifecycle of the status of your files.

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.

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.

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
#
```

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.

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 --staged
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 --staged` to see what you've staged.

```
$ git diff --staged
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
```

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.

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.

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 ~.

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.

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 co\
mmand.
--name-only        Show the list of files modified after the commit information.
--name-status        Show the list of files affected with added/modified/deleted informa\
tion 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 date format.
--graph        Display an ASCII graph of the branch and merge history beside the log out\
put.
--pretty        Show commits in an alternate format. Options include oneline, short, ful\
l, fuller, and format (where you specify your own format).
--oneline        A convenience option short for `--pretty=oneline --abbrev-commit`.
```

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.
```

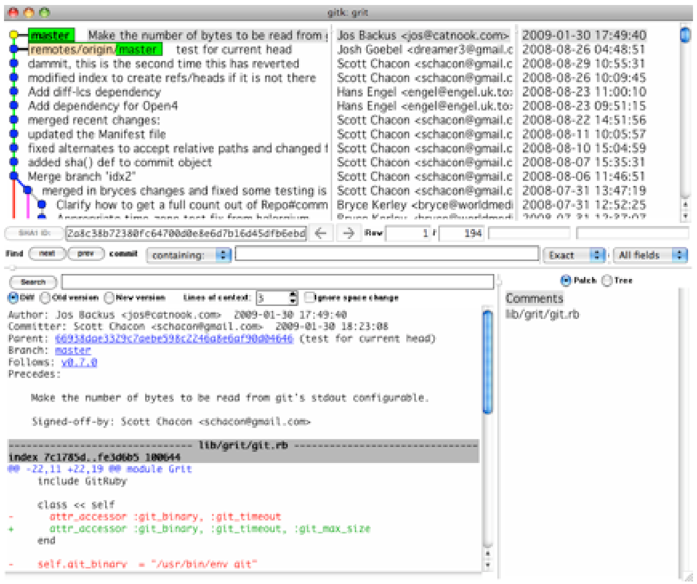
For example, to see which commits modifying test files in the Git source code history were committed by Junio Hamano in the month of October 2008 and were not merges, run something like

```
$ 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.

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.



```
Make the number of bytes to be read from g   Jos Backus <jos@catnook.com>     2009-01-30 17:49:40
test for current head                        Josh Goebel <dreamer3@gmail.c    2008-08-26 04:48:51
dammit, this is the second time this has reverted   Scott Chacon <schacon@gmail.c   2008-08-29 10:55:31
modified index to create refs/heads if it is not there   Scott Chacon <schacon@gmail.c   2008-08-26 10:09:45
Add diff-lcs dependency                      Hans Engel <engel@engel.uk.to:   2008-08-23 11:00:10
Add dependency for Open4                     Hans Engel <engel@engel.uk.to:   2008-08-23 09:51:15
merged recent changes:                       Scott Chacon <schacon@gmail.c    2008-08-22 14:51:56
updated the Manifest file                    Scott Chacon <schacon@gmail.c    2008-08-11 10:05:57
fixed alternates to accept relative paths and changed t   Scott Chacon <schacon@gmail.c   2008-08-10 15:04:59
added sha() def to commit object             Scott Chacon <schacon@gmail.c    2008-08-07 15:35:31
Merge branch 'idx2'                          Scott Chacon <schacon@gmail.c    2008-08-06 11:46:51
merged in bryces changes and fixed some testing is   Scott Chacon <schacon@gmail.c   2008-07-31 13:47:19
Clarify how to get a full count out of Repo#comm   Bryce Kerley <bryce@worldmedi   2008-07-31 12:52:25

2a8c38b72380fc64700d0e8e6d7b16d45dfb6ebd   Row 1 / 194

Author: Jos Backus <jos@catnook.com>  2009-01-30 17:49:40
Committer: Scott Chacon <schacon@gmail.com>  2009-01-30 18:23:08
Parent: 66938dae3329c7aebe598c2246a8e6af90d04646 (test for current head)
Branch: master
Follows: v0.7.0
Precedes:

    Make the number of bytes to be read from git's stdout configurable.

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

------------------------------ lib/grit/git.rb ------------------------------
index 7c1785d..fe3d6b5 100644
@@ -22,11 +22,19 @@ module Grit
   include GitRuby

   class << self
-    attr_accessor :git_binary, :git_timeout
+    attr_accessor :git_binary, :git_timeout, :git_max_size
   end

-  self.git_binary  = "/usr/bin/env git"
```

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.

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.

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.

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.

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.

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.

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*).

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*.)

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*.

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.

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.

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
```

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.

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
```

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.

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'
```

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.

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'
```

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'
```

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
...
```

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.

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.

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.

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'
```

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.

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.

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.

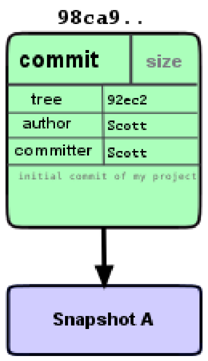


Figure 3-1. Single commit repository data.

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.

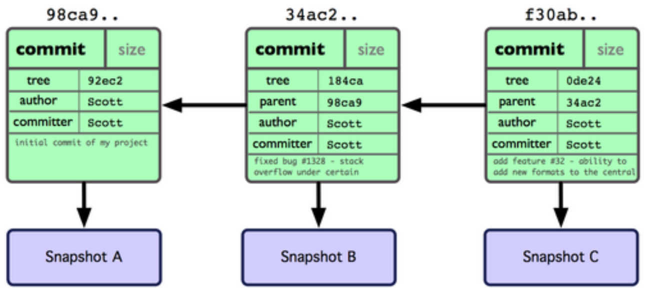


Figure 3-2. Git object data for multiple commits.

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.

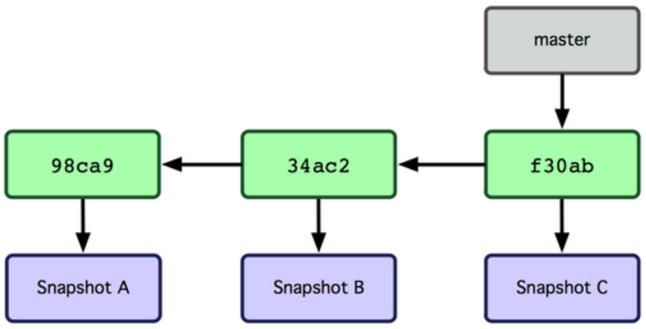


Figure 3-3. Branch pointing into the commit history.

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`.

```
$ 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).

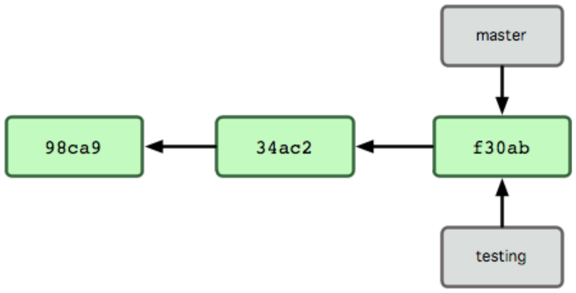


Figure 3-4. Multiple branches pointing into the commit's data history.

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`.

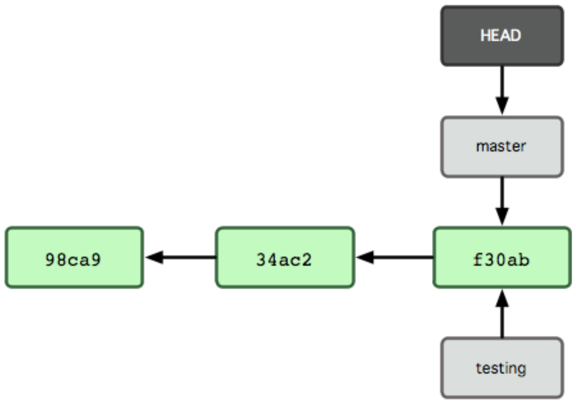


Figure 3-5. HEAD file pointing to the branch you're on.

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).

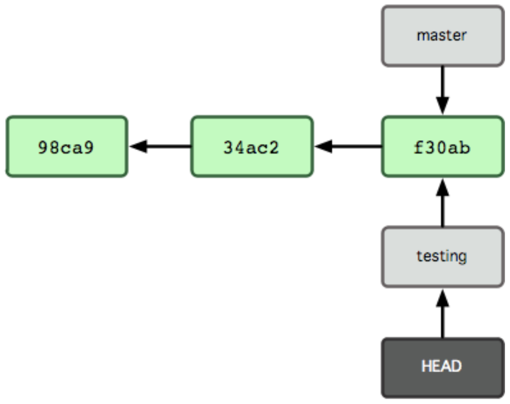


Figure 3-6. HEAD points to a different branch when you switch branches.

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.

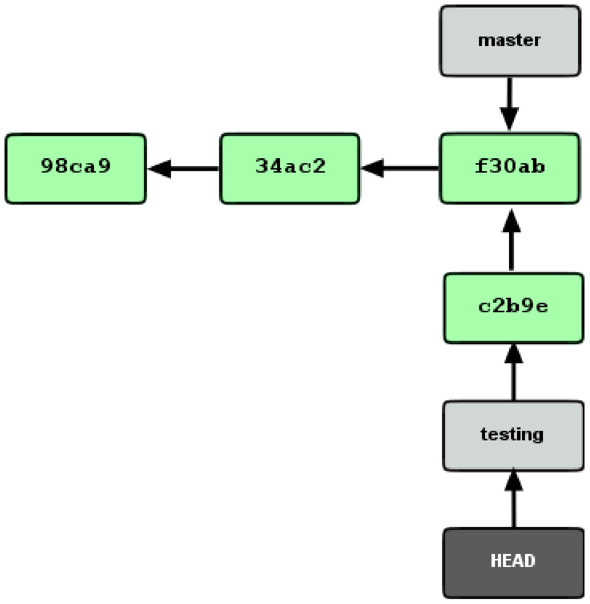


Figure 3-7. The branch that HEAD points to moves forward with each commit.

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.

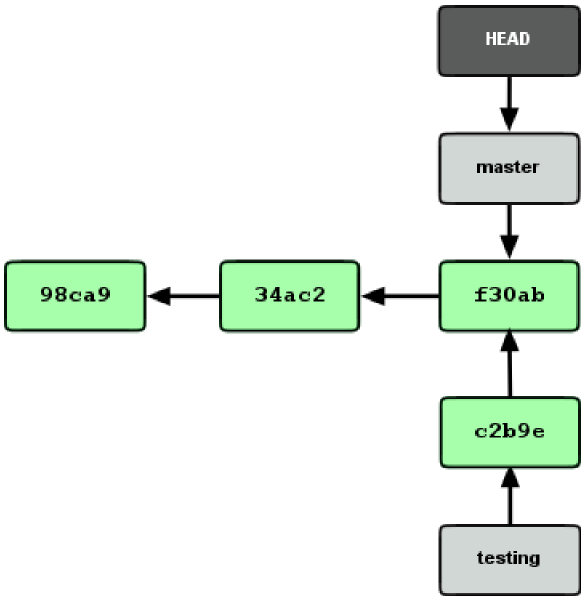


Figure 3-8. HEAD moves to another branch on a checkout.

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.

```
$ 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.

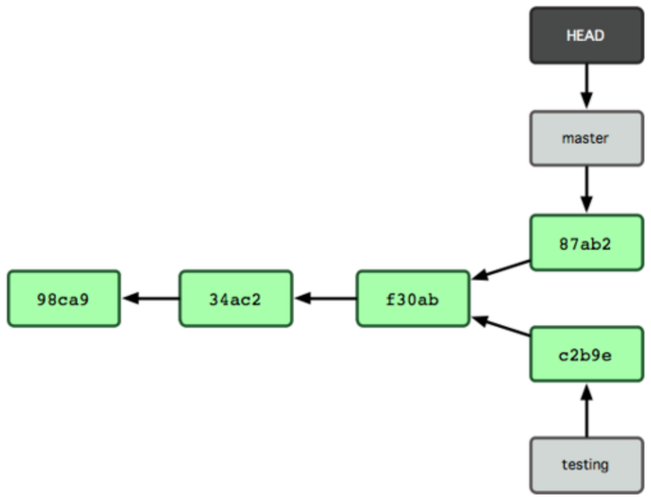


Figure 3-9. The branch histories have diverged.

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.

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.

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).

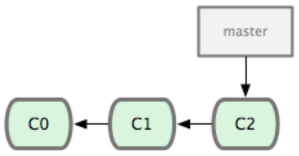


Figure 3-10. A short and simple commit history.

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.

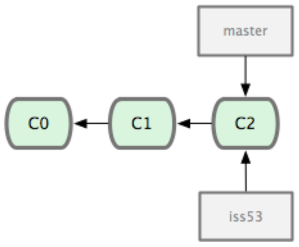


Figure 3-11. Creating a new branch pointer.

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]'
```

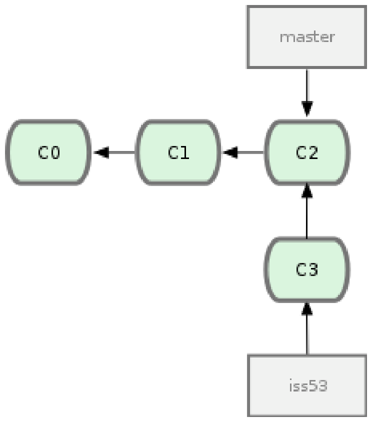


Figure 3-12. The iss53 branch has moved forward with your work.

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:

```
$ 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(-)
```

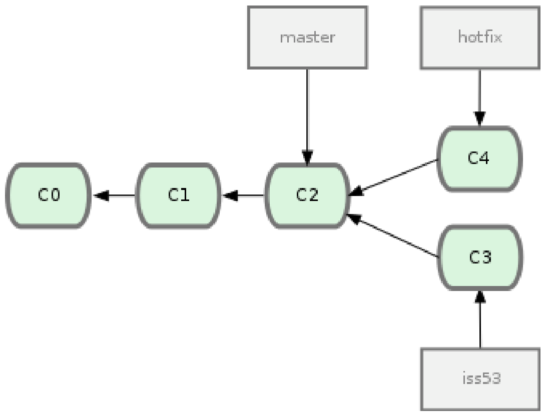


Figure 3-13. hotfix branch based back at your master branch point.

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).

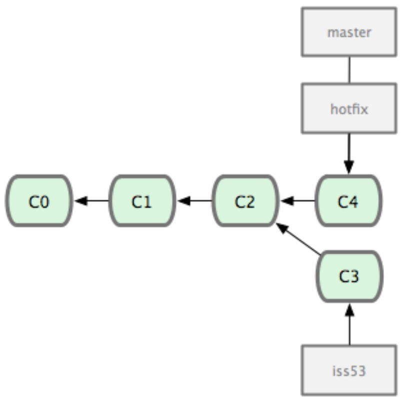


Figure 3-14. Your master branch points to the same place as your hotfix branch after the merge.

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`.

```
$ 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(-)
```

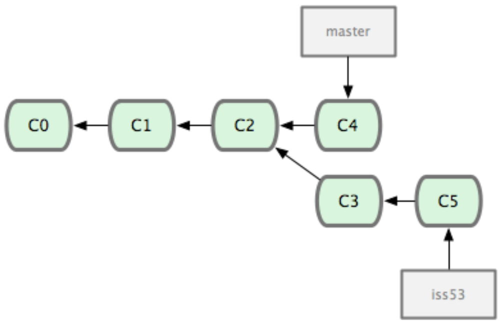


Figure 3-15. Your iss53 branch can move forward independently.

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.

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.

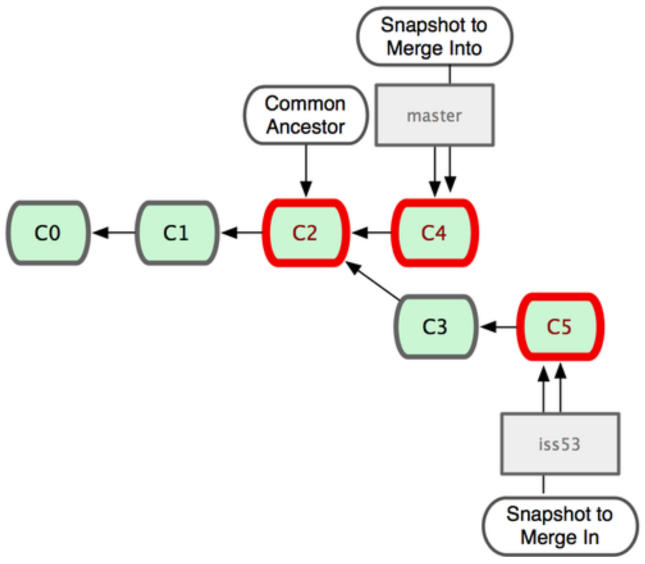


Figure 3-16. Git automatically identifies the best common-ancestor merge base for branch merging.

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.

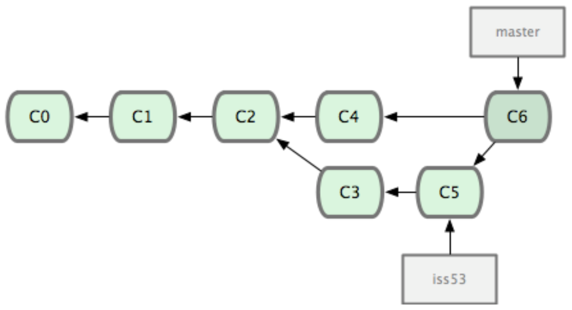


Figure 3-17. Git automatically creates a new commit object that contains the merged work.

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
```

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.

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.

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.

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).

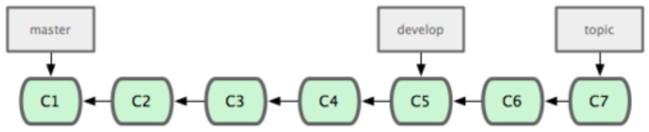


Figure 3-18. More stable branches are generally farther to the left in commit history.

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).

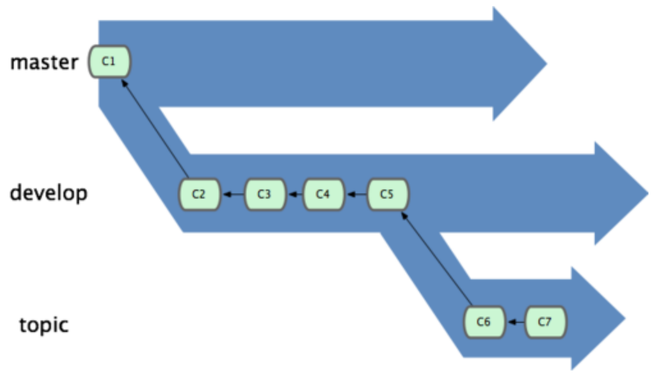


Figure 3-19. It may be helpful to think of your branches as silos.

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.

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.

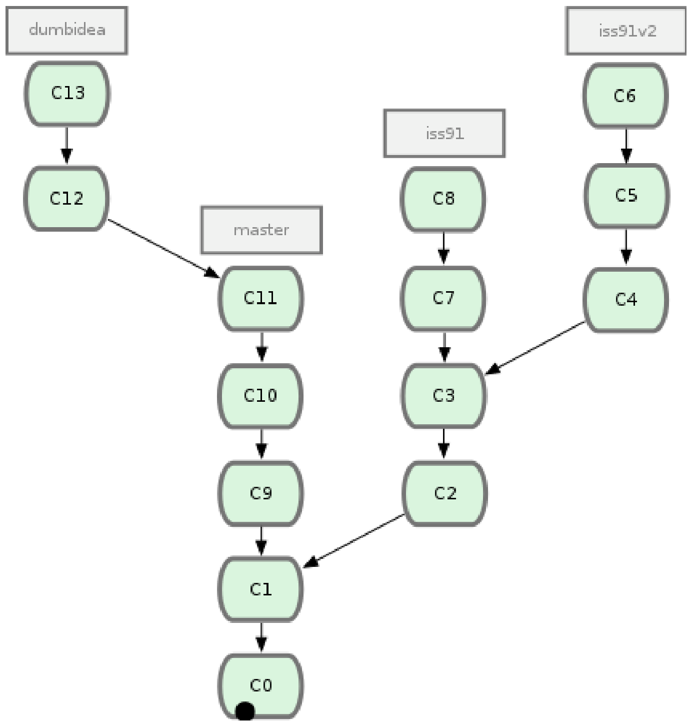


Figure 3-20. Your commit history with multiple topic branches.

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.

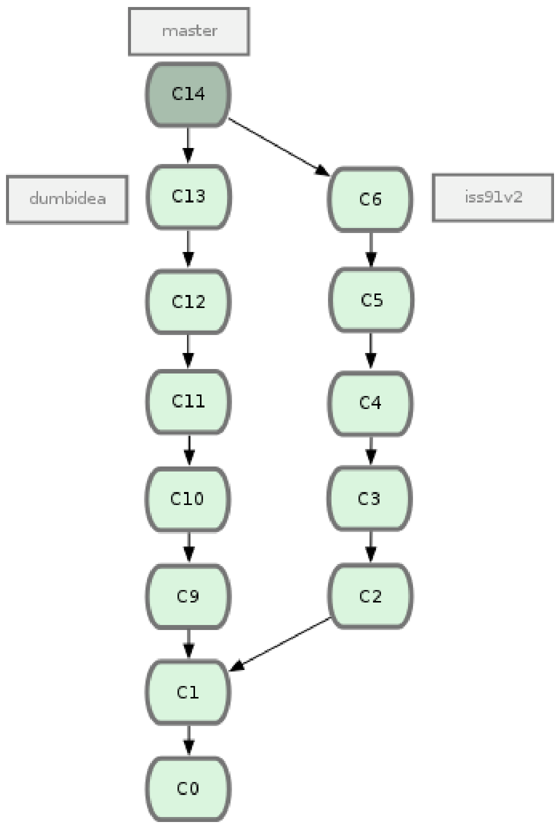


Figure 3-21. Your history after merging in dumbidea and iss91v2.

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.

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).

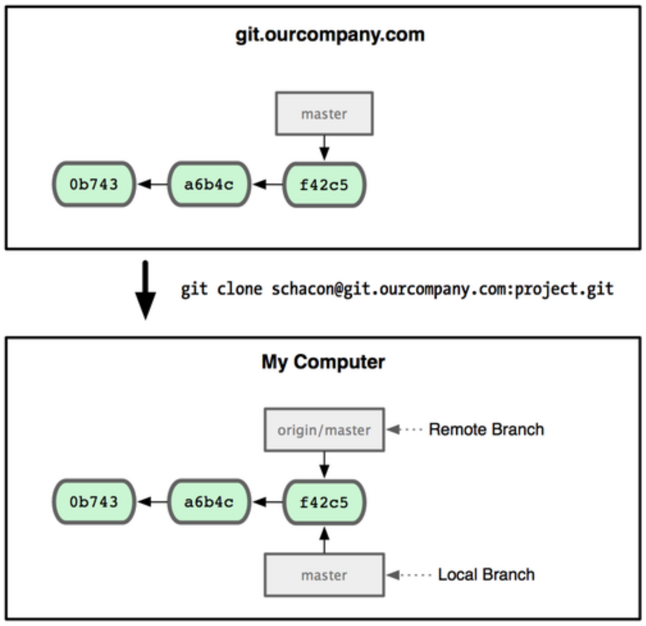


Figure 3-22. A Git clone gives you your own master branch and origin/master pointing to origin's master branch.

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).

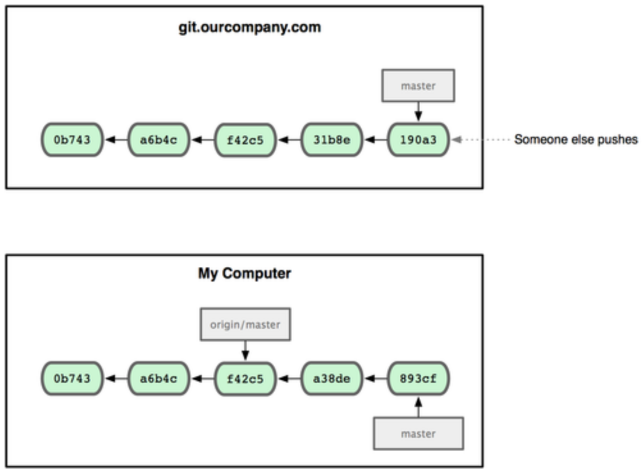


Figure 3-23. Working locally and having someone push to your remote server makes each history move forward differently.

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).

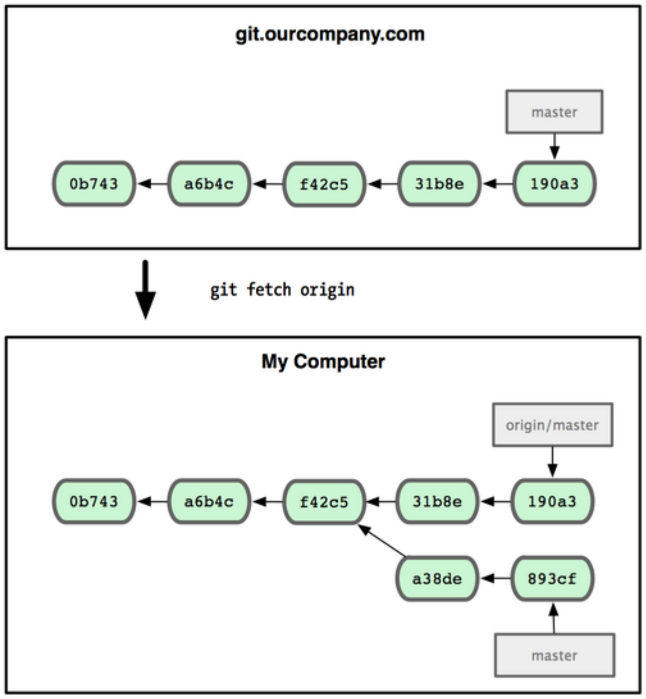


Figure 3-24. The git fetch command updates your remote references.

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).

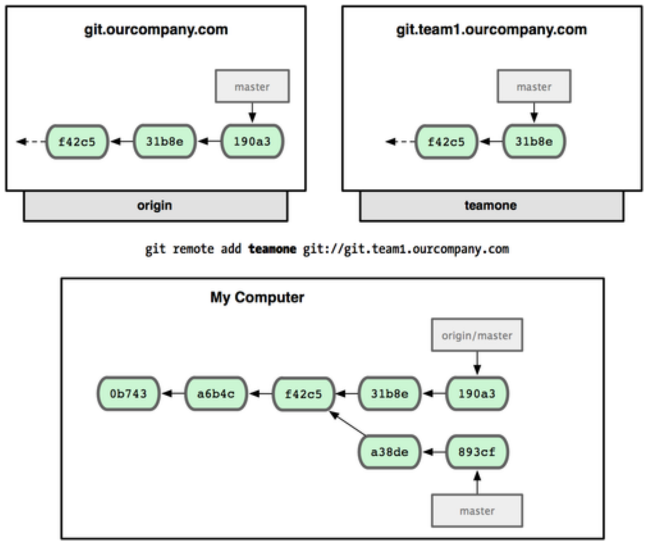


Figure 3-25. Adding another server as a remote.

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).

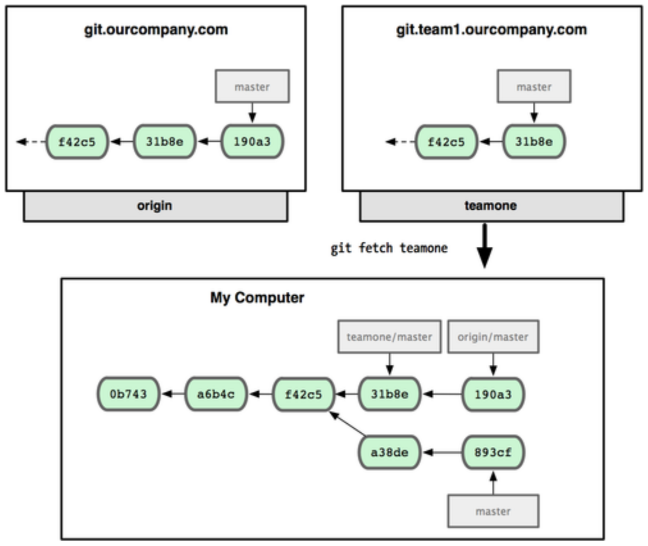


Figure 3-26. You get a reference to teamone's master branch position locally.

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.

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`.

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]`".

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.

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.

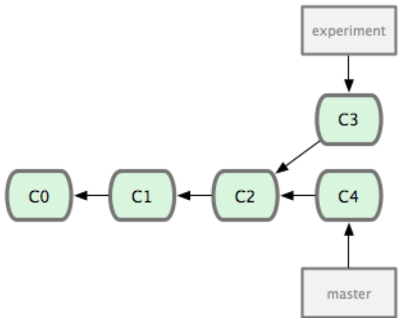


Figure 3-27. Your initial diverged commit history.

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.

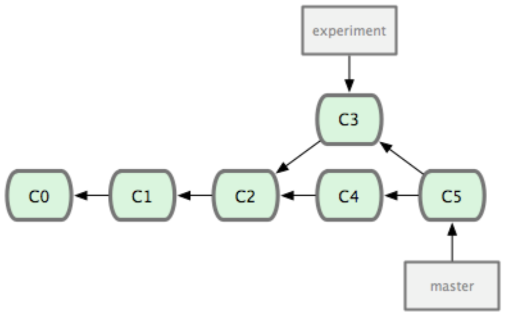


Figure 3-28. Merging a branch to integrate the diverged work history.

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

```
$ 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.

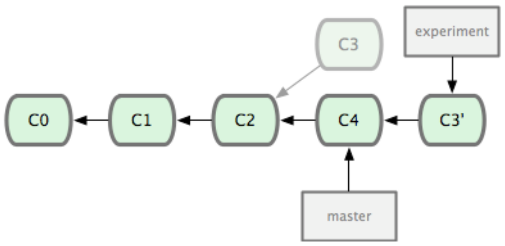


Figure 3-29. Rebasing the change introduced in C3 onto C4.

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

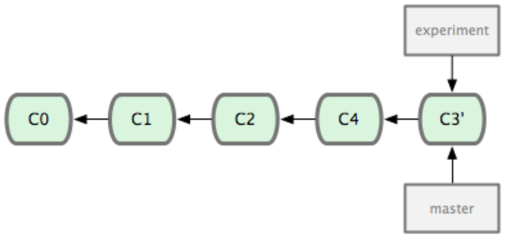


Figure 3-30. Fast-forwarding the master branch.

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

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

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

More Interesting Rebases

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).

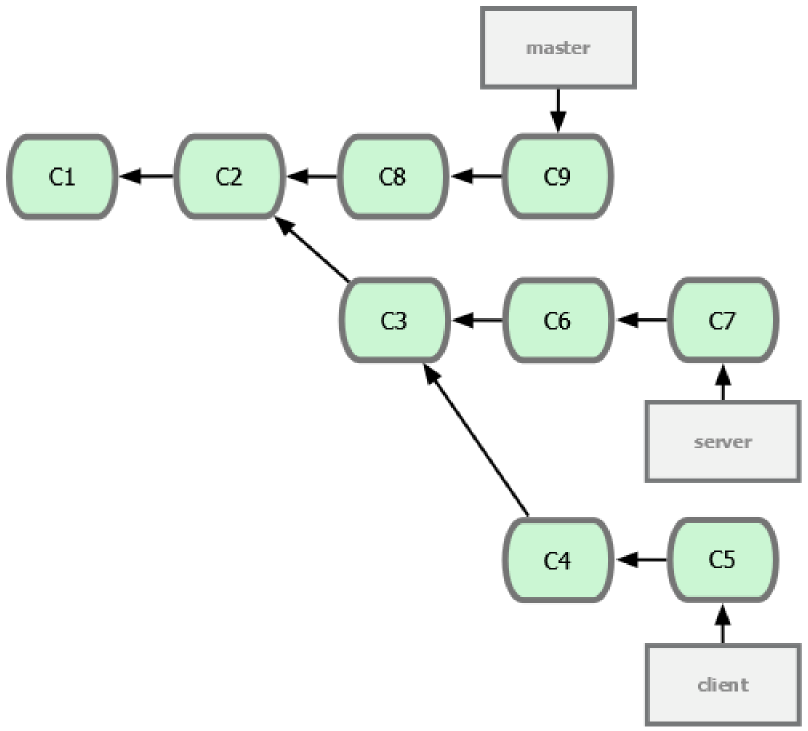


Figure 3-31. A history with a topic branch off another topic branch.

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.

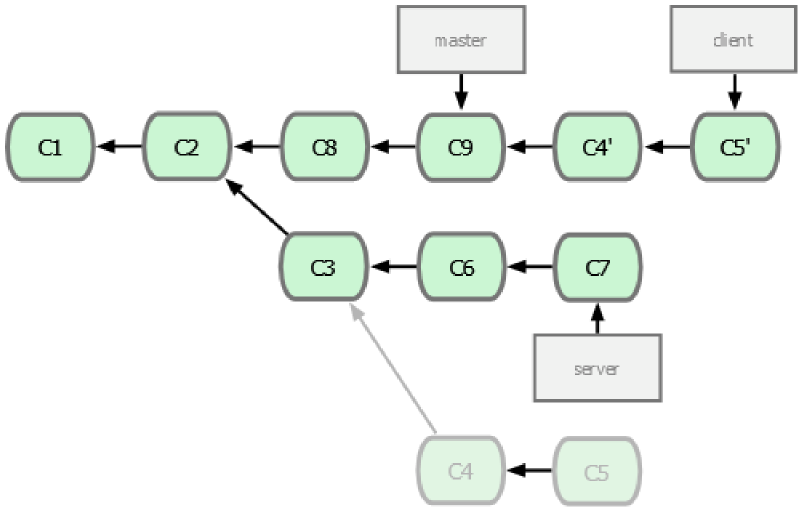


Figure 3-32. Rebasing a topic branch off another topic branch.

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

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

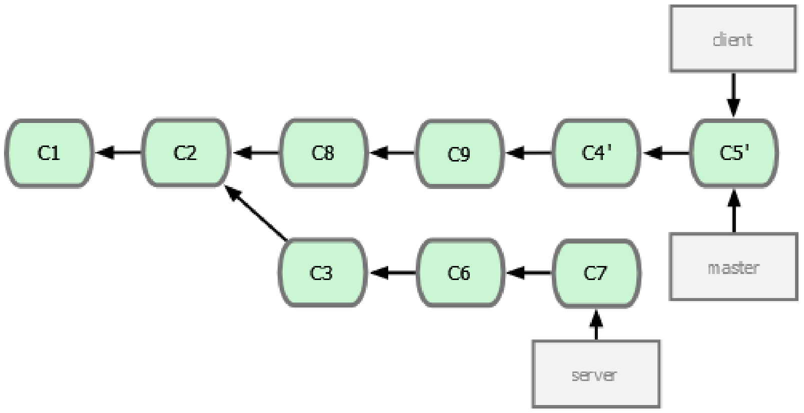


Figure 3-33. Fast-forwarding your master branch to include the client branch changes.

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.

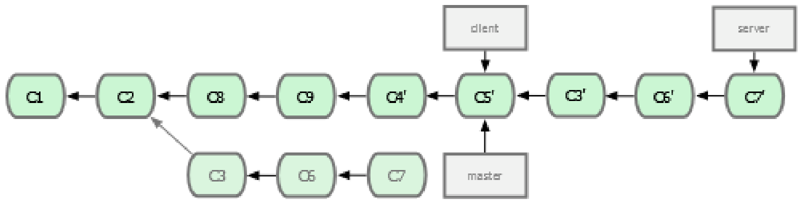


Figure 3-34. Rebasing your server branch on top of your master branch.

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
```

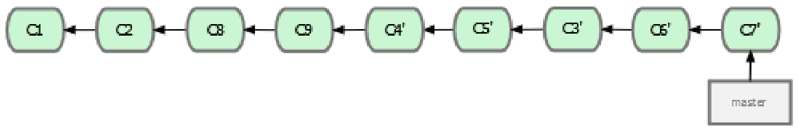


Figure 3-35. Final commit history.

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.

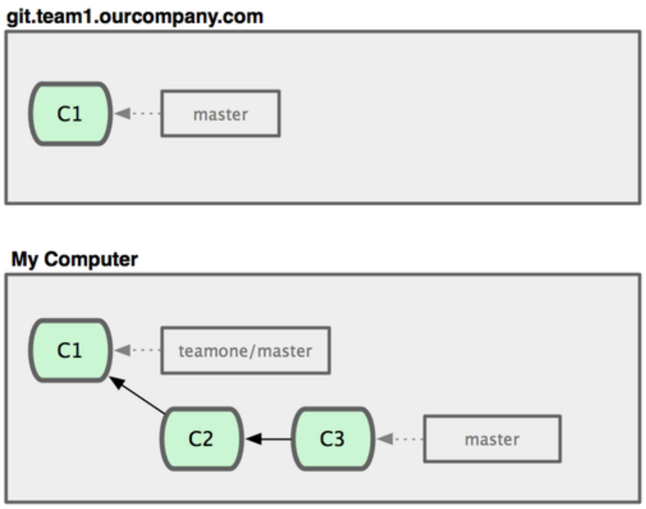


Figure 3-36. Clone a repository, and base some work on it.

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.

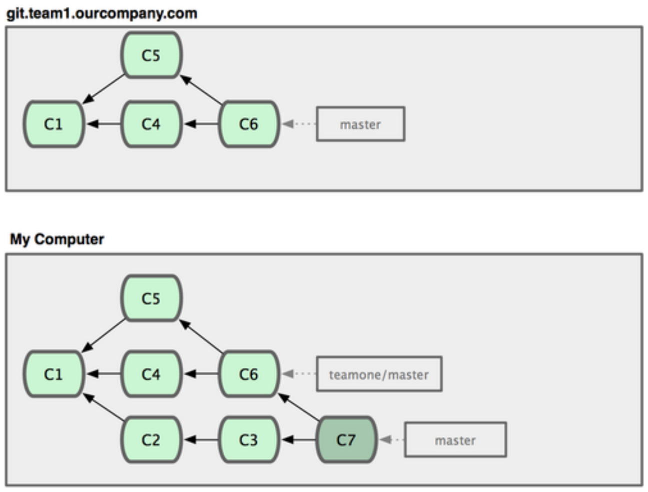


Figure 3-37. Fetch more commits, and merge them into your work.

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.

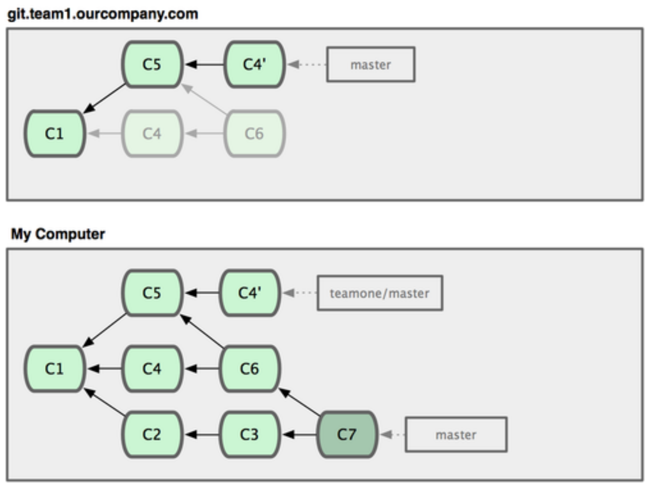


Figure 3-38. Someone pushes rebased commits, abandoning commits you've based your work on.

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).

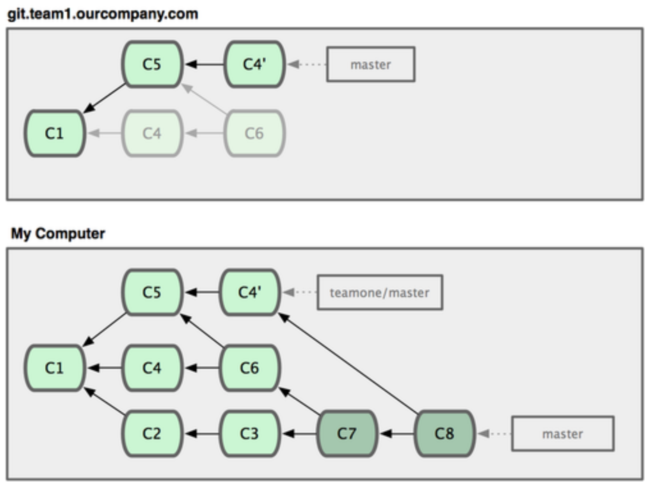


Figure 3-39. You merge in the same work again into a new merge commit.

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.

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.