A PDF document generator

Panbdoc Matt Smith

May 2017

Contents

Chapter 1

Getting Started

This chapter will be about getting started with Git. We will begin by explaining some background on version control tools, then move on to how to get Git running on your system and finally how to get it set up to start working with. At the end of this chapter you should understand why Git is around, why you should use it and you should be all set up to do so.

See Smith's book (2015).

Others have also written nice books (Adams 1979).

1.1 Footnote examples

Here is a footnote reference,[1] and another.[2]

This paragraph won't be part of the note, because it isn't indented.

1.2 About Version Control

What is version control, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. Even though the examples in this book show software

[1] Here is the footnote.

[2] Here's one with multiple blocks. Subsequent paragraphs are indented to show that they belong to the previous footnote.

```
{ some.code }
```

The whole paragraph can be indented, or just the first line. In this way, multi-paragraph footnotes work like multi-paragraph list items.

source code as the files under version control, in reality any type of file on a computer can be placed under version control.

If you are a graphic or web designer and want to keep every version of an image or layout (which you certainly would), it is very wise to use a Version Control System (VCS). A VCS allows you to: revert files back to a previous state, revert the entire project back to a previous state, review changes made over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also means that if you screw things up or lose files, you can generally recover easily. In addition, you get all this for very little overhead.

1.2.1 Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control (see Figure 1-1).

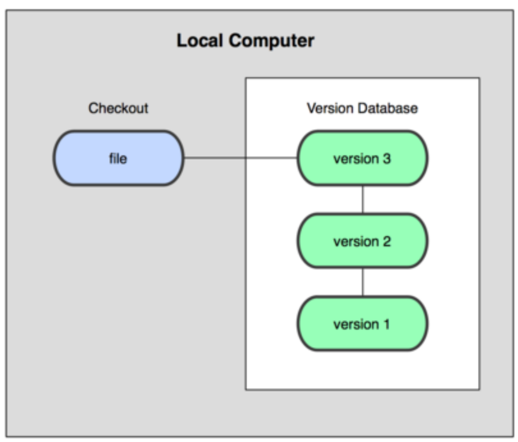

Figure 1.1: Local version control diagram.

One of the more popular VCS tools was a system called rcs, which is still distributed with many computers today. Even the popular Mac OS X operating system includes the rcs command 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 in a special format on disk; it can then recreate what any file looked like at any point in time by adding up all the patches.

1.2.2 Centralized Version Control Systems

The next major issue that people encounter is that they need 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, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control (see Figure 1-2).

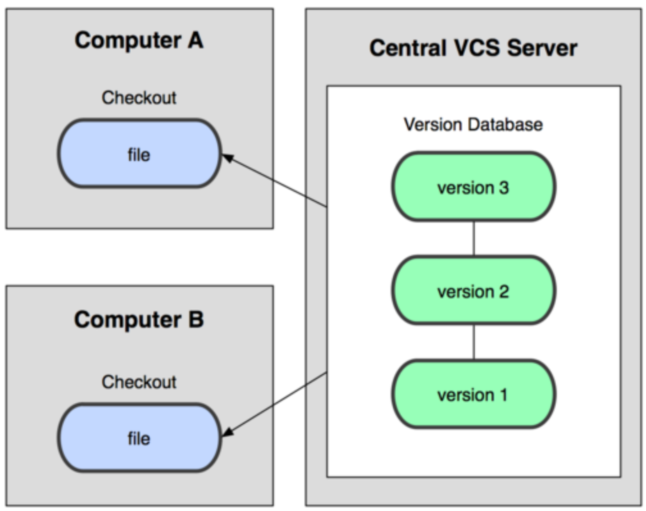

Figure 1.2: Centralized version control diagram.

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators 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 databases on every client.

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

1.2.3 Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every checkout is really a full backup of all the data (see Figure 1-3).

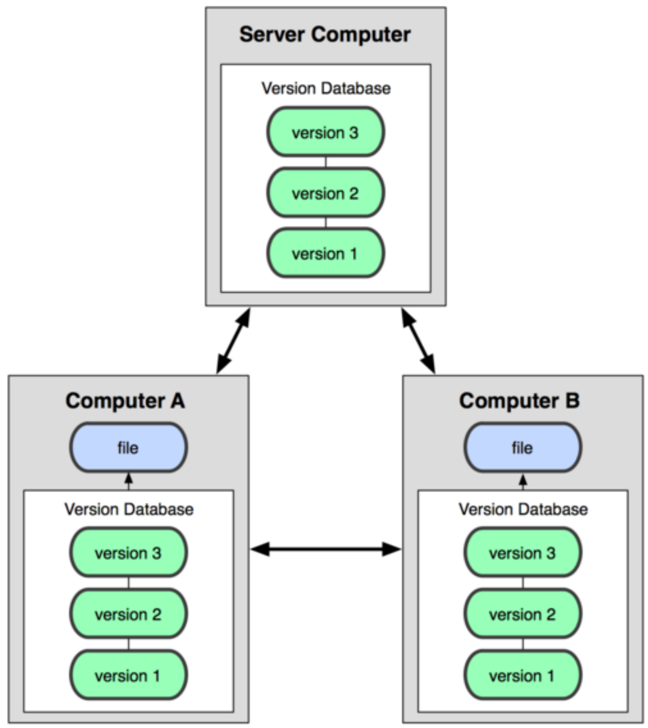

Figure 1.3: Distributed version control diagram.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

1.3 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 open source software project of fairly large scope. For most of the lifetime of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files. In 2002, the Linux kernel project began using a proprietary DVCS system called BitKeeper.

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were as follows:

- 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 to be easy to use and yet retain these initial qualities. It's incredibly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development (See Chapter 3).

1.4 Git Basics

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

1.4.1 Snapshots, Not Differences

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time, as illustrated in Figure 1-4.

Git doesn't think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a mini filesystem. Every time you commit, or

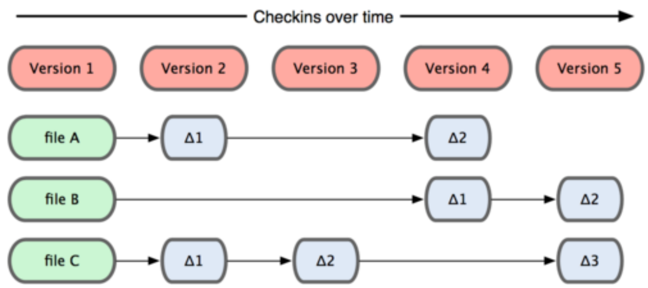

Figure 1.4: Other systems tend to store data as changes to a base version of each file.

save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again—just a link to the previous identical file it has already stored. Git thinks about its data more like Figure 1-5.

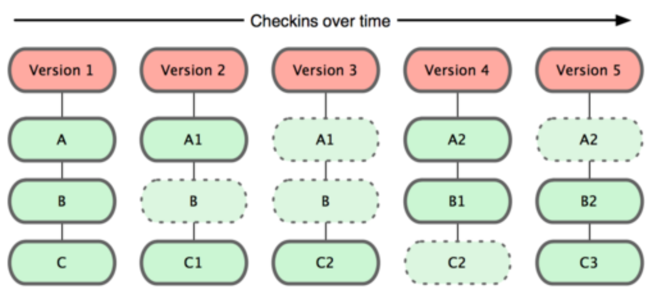

Figure 1.5: Git stores data as snapshots of the project over time.

This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation. This makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS. We'll explore some of the benefits you gain by thinking of your data this way when we cover Git branching in Chapter 3.

1.4.2 Nearly Every Operation Is Local

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

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

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

1.4.3 Git Has Integrity

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

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

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

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

1.4.4 Git Generally Only Adds Data

When you do actions in Git, nearly all of them only add data to the Git database. It is very difficult to get the system to do anything that is not undoable or to make it erase 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 snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

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

1.4.5 The Three States

Now, pay attention. This is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: committed, modified, and staged. Committed means that the data is safely stored in your local database. Modified means that you have changed the file but have not committed it to your database yet. Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

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

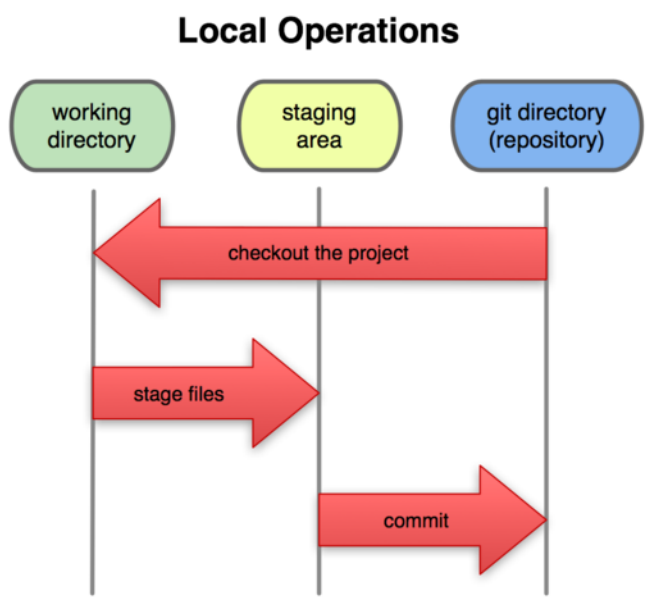

Figure 1.6: Working directory, staging area, and Git directory.

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

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

The staging area is a simple file, generally contained in your Git directory, that stores information about 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 basic Git workflow goes something like this:

1. You modify files in your working directory.
2. You stage the files, adding snapshots of them to your staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

If a particular version of a file is in the Git directory, it's considered committed. If it's modified but has been added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified. In Chapter 2, you'll learn more about these states and how you can either take advantage of them or skip the staged part entirely.

1.5 Installing Git

Let's get into using some Git. First things first—you have to install it. You can get it a number of ways; the two major ones are to install it from source or to install an existing package for your platform.

1.5.1 Installing from Source

If you can, it's generally useful to install Git from source, because you'll get the most recent version. Each version of Git tends to include useful UI enhancements, so getting the latest version is often the best route if you feel comfortable compiling software from source. It is also the case that many Linux distributions contain very old packages; so unless you're on a very up-to-date distro or are using backports, installing from source may be the 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 apt-get (such as a Debian based system), you can use one of these commands to install all of the dependencies:

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

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

When you have all the necessary dependencies, you can go ahead and grab the latest snapshot from the Git web site:

http://git-scm.com/download

Then, compile and install:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

After this is done, you can also get Git via Git itself for updates:

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

1.5.2 Installing on Linux

If you want to install Git on Linux via a binary installer, you can generally do so through the basic package-management tool that comes with your distribution. If you're on Fedora, you can use yum:

```
$ yum install git
```

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

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

1.5.3 Installing on Mac

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

http://sourceforge.net/projects/git-osx-installer/

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

```
$ sudo port install git +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).

Homebrew (`http://brew.sh/`) is another alternative to install Git. If you have Homebrew installed, install Git via

```
$ brew install git
```

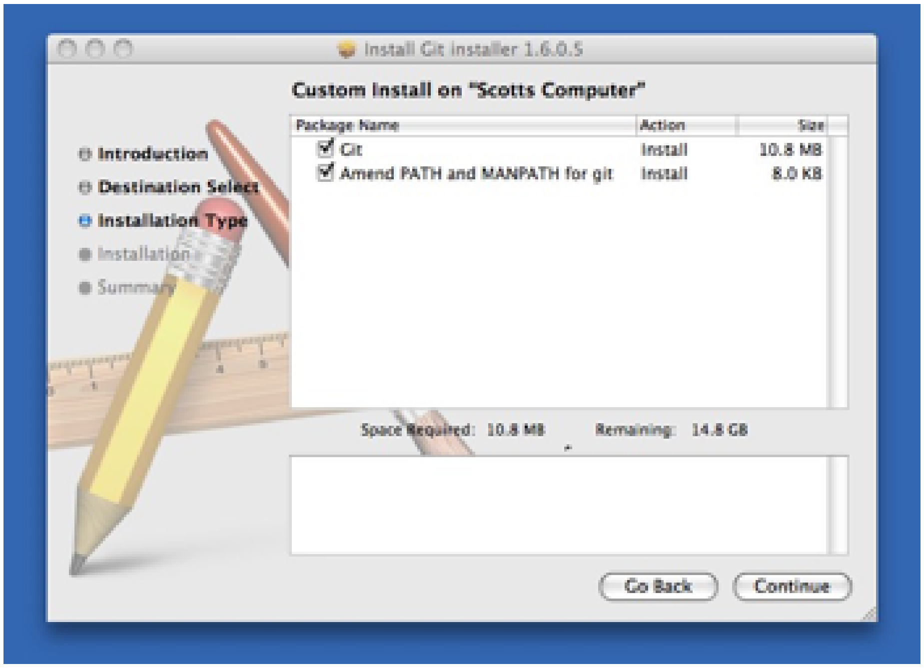

Figure 1.7: Git OS X installer.

1.5.4 Installing on Windows

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

`http://msysgit.github.io`

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

Note on Windows usage: you should use Git with the provided msysGit shell (Unix style), it allows to use the complex lines of command given in this book. If you need, for some reason, to use the native Windows shell / command line console, you have to use double quotes instead of single quotes (for parameters with spaces in them) and you must quote the parameters ending with the circumflex accent (^) if they are last on the line, as it is a continuation symbol in Windows.

1.6 First-Time Git Setup

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

Git comes with a tool called `git config` that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

- `/etc/gitconfig` file: Contains values for every user on the system and all their repositories. If you pass the option`--system` to `git config`, it reads and writes from this file specifically.
- `~/.gitconfig` file: Specific to your user. You can make Git read and write to this file specifically by passing the `--global` option.
- config file in the Git directory (that is, `.git/config`) of whatever repository you're currently using: Specific to that single repository. Each level overrides values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`.

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

1.6.1 Your Identity

The first thing you should do when you install Git is to set your user name and e-mail address. This is important because every Git commit uses this information, and 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 need to do this only once if you pass the `--global` option, because then Git will always use that information for anything you do on that system. If you want to override this with a different name or e-mail address for specific projects, you can run the command without the `--global` option when you're in that project.

1.6.2 Your Editor

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. By default, Git uses your system's default editor, which is generally Vi or Vim. If you want to use a different text editor, such as Emacs, you can do the following:

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

1.6.3 Your Diff Tool

Another useful option you may want to configure is the default diff tool to use to resolve merge conflicts. Say you want to use vimdiff:

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

Git accepts kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, and opendiff as valid merge tools. You can also set up a custom tool; see Chapter 7 for more information about doing that.

1.6.4 Checking Your Settings

If you want to check your settings, you can use the `git config --list` command to list all the settings Git can find at that point:

```
$ 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 reads the same key from different files (`/etc/gitconfig` and `~/.gitconfig`, for example). In this case, Git uses the last value for each unique key it sees.

You can also check what Git thinks a specific key's value is by typing `git config {key}`:

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

1.7 Getting Help

If you ever need help while using Git, there are three ways to get the manual page (manpage) help for any of the Git commands:

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

For example, you can get the manpage help for the config command by running

```
$ git help config
```

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

1.8 Summary

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

Chapter 2

Git Basics

If you can read only one chapter to get going with Git, this is it. 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. We'll also show you how to set up Git to ignore certain files and file patterns, how to undo mistakes quickly and easily, how to browse the history of your project and view changes between commits, and how to push and pull from remote repositories.

2.1 Getting a Git Repository

You can get a Git project using two main approaches. The first takes an existing project or directory and imports it into Git. The second clones an existing Git repository from another server.

2.1.1 Initializing a Repository in an Existing Directory

If you're starting to track an existing project in Git, you need to go to the project's directory and type

```
$ git init
```

This creates a new subdirectory named `.git` that contains all of your necessary repository files — a Git repository skeleton. At this point, nothing in your project is tracked yet. (See *Chapter 9* for more information about exactly what files are contained in the `.git` directory you just created.)

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

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

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

2.1.2 Cloning an Existing Repository

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

You clone a repository with `git clone [url]`. For example, if you want to clone the Ruby Git library called Grit, you can do so like this:

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

That creates a directory named `grit`, initializes a `.git` directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new `grit` directory, you'll see the project files in there, ready to be worked on or used. If you want to clone the repository into a directory named something other than grit, you can specify that as the next command-line option:

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

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

Git has a number of different transfer protocols you can use. The previous example uses the `git://` protocol, but you may also see `http(s)://` or `user@server:/path.git`, which uses the SSH transfer protocol. *Chapter 4* will introduce all of the available options the server can set up to access your Git repository and the pros and cons of each.

2.2 Recording Changes to the Repository

You have a bona fide Git repository and a checkout or working copy of the files for that project. You need to make some changes and commit snapshots of those changes into your repository each time the project reaches a state you want to record.

Remember that each file in your working directory can be in one of two states: *tracked* or *untracked*. *Tracked* files are files that were in the last snapshot; they can be *unmodified*, *modified*, or *staged*. *Untracked* files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because you just checked them out and haven't edited anything.

As you edit files, Git sees them as modified, because you've changed them since your last commit. You *stage* these modified files and then commit all your staged changes, and the cycle repeats. This lifecycle is illustrated in Figure 2-1.

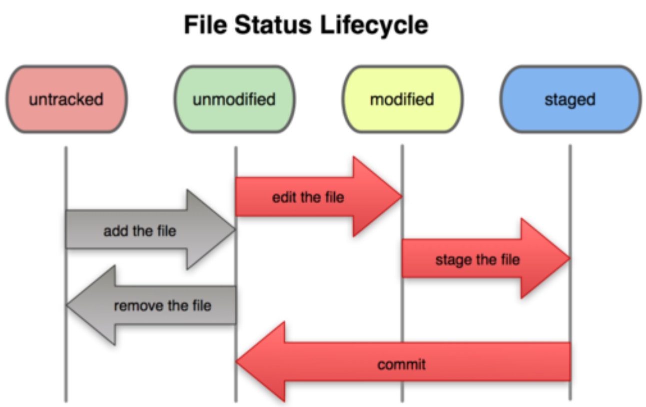

Figure 2.1: The lifecycle of the status of your files.

2.2.1 Checking the Status of Your Files

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

```
$ git status
On branch master
nothing to commit, working directory clean
```

This means you have a clean working directory — in other words, no tracked files are modified. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tells you which branch you're on. For now, that is always `master`, which is the default; you won't worry about it here. The next chapter will go over branches and references in detail.

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

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

2.2.2 Tracking New Files

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

```
$ git add README
```

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

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

        new file:   README
```

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

2.2.3 Staging Modified Files

Let's change a file that was already tracked. If you change a previously tracked file called `benchmarks.rb` and then run your `status` command again, you get something that looks like this:

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

        modified:   benchmarks.rb
```

The `benchmarks.rb` file appears under a section named "Changes not staged for commit" — which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the `git add` command (it's a multipurpose command — you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved). Let's run `git add` now to stage the `benchmarks.rb` file, 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 in `benchmarks.rb` before you commit it. You open it again and make that change, and you're ready to commit. However, let's run `git status` one more time:

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

        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 stages a file exactly as it is when you run the `git add` command. If you commit now, the version of `benchmarks.rb` as it was when you last ran the `git add` command is how it will go into the commit, not the version of the file as it looks in your working directory when you run `git commit`. If you modify a file after you run `git add`, you have to run `git add` again to stage the latest version of the file:

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

        new file:   README
        modified:   benchmarks.rb
```

2.2.4 Ignoring Files

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

```
$ cat .gitignore
*.[oa]
*~
```

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

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

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

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

Here is another example `.gitignore` file:

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

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

2.2.5 Viewing Your Staged and Unstaged Changes

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

Let's say you edit and stage the `README` file again and then edit the `benchmarks.rb` file without staging it. If you run your `status` command, you once again see something like this:

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

        modified:   benchmarks.rb
```

To see what you've changed but not yet staged, type `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 is in your working directory with what is in your staging area. The result tells you the changes you've made that you haven't yet staged.

If you want to see what you've staged that will go into your next commit, you can use `git diff --cached`. (In Git versions 1.6.1 and later, you can also use `git diff --staged`, which may be easier to remember.) This command compares your staged changes to your last commit:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
```

```
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

It's important to note that `git diff` by itself doesn't show all changes made since your last commit — only changes that are still unstaged. This can be confusing, because if you've staged all of your changes, `git diff` will give you no output.

For another example, if you stage the `benchmarks.rb` file and then edit it, you can use `git diff` to see the changes in the file that are staged and the changes that are unstaged:

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   benchmarks.rb

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
```

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

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

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

and `git diff --cached` to see what you've staged so far:

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
```

```
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
          @commit.parents[0].parents[0].parents[0]
        end

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

2.2.6 Committing Your Changes

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

```
$ git commit
```

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

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

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       new file:   README
#       modified:   benchmarks.rb
#
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

You can see that the default commit message contains the latest output of the `git status` command commented out and one empty line on top. You can remove these comments and type your commit message, or you can leave them there to help you remember what you're committing. (For an even more explicit reminder of what you've modified, you can pass the `-v` option to `git commit`. Doing so also puts the diff of your change in the editor so you can see exactly what you did.) When you exit the editor, Git creates your commit with that commit message (with the comments and diff stripped out).

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

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

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

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

2.2.7 Skipping the Staging Area

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

```
$ git status
On branch master
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

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 files changed, 5 insertions(+)
```

Notice how you don't have to run `git add` on the `benchmarks.rb` file in this case before you commit.

2.2.8 Removing Files

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

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

```
$ rm grit.gemspec
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    grit.gemspec

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

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

```
$ git rm 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 the file and added it to the index already, you must force the removal with the `-f` option. This is a safety feature to prevent accidental removal of data that hasn't yet been recorded in a snapshot and that can't be recovered from Git.

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

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

You can pass files, directories, and file-glob patterns to the `git rm` command.

That means you can do things such as

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

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

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

This command removes all files that end with `~`.

2.2.9 Moving Files

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

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

```
$ git mv file_from file_to
```

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

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

        renamed:    README -> README.txt
```

However, this is equivalent to running something like this:

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

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

2.3 Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool 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 this:

```
$ 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 the commits made in that repository in reverse chronological order. That is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author's name and e-mail, the date written, and the commit message.

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

One of the more helpful options is `-p`, which shows the diff introduced in each commit. You can also use `-2`, 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 option displays the same information but with a diff directly following each entry. This is very helpful for code review or to quickly browse what happened during a series of commits that a collaborator has added.

Sometimes it's easier to review changes on the word level rather than on the line level. There is a `--word-diff` option available in Git, that you can append to the `git log -p` command to get word diff instead of normal line by line diff. Word diff format is quite useless when applied to source code, but it comes in handy when applied to large text files, like books or your dissertation. Here is 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 is no added and removed lines in this output as in a normal diff. Changes are shown inline instead. You can see the added word enclosed in `{+ +}` and removed one enclosed in `[- -]`. You may also want to reduce the usual three lines context in diff output to only one line, as the context is now words, not lines. You can do this with `-U1` as we did in the example above.

You can also use a series of summarizing options with `git log`. For example, if you want to see some abbreviated stats for each commit, you can use the `--stat` option:

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

    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, 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(+)
```

As you can see, the `--stat` option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end. Another really useful option is `--pretty`. This option changes the log output to formats other than the default. A few prebuilt options are available for you to use. The `oneline` option prints each commit on a single line, which is useful if you're looking at a lot of commits. In addition, the `short`, `full`, and `fuller` 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 machine parsing — because you specify the format explicitly, you know it won't change with updates to Git:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 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 takes.

```
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. We'll cover this distinction a bit more in *Chapter 5*.

The `oneline` and `format` options are particularly useful with another `log` option called `--graph`. This option adds a nice little ASCII graph showing your branch and merge history, which we can see in our 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 we've covered so far and some other common formatting options that may be useful, along with how they change the output of the `log` command.

```
Option  Description
-p  Show the patch introduced with each commit.
--word-diff Show the patch in a word diff format.
--stat  Show statistics for files modified in each commit.
--shortstat Display only the changed/insertions/deletions line from the --stat command
--name-only Show the list of files modified after the commit information.
--name-status   Show the list of files affected with added/modified/deleted informatio
--abbrev-commit Show only the first few characters of the SHA-1 checksum instead of al
--relative-date Display the date in a relative format (for example, "2 weeks ago") ins
--graph Display an ASCII graph of the branch and merge history beside the log output.
--pretty    Show commits in an alternate format. Options include oneline, short, full,
--oneline   A convenience option short for `--pretty=oneline --abbrev-commit`.
```

2.3.1 Limiting Log Output

In addition to output-formatting options, `git log` takes a number of useful limiting options — that is, options that let you show only 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 do `-<n>`, where `n` is any integer to show the last `n` commits. In reality, you're unlikely to use that often, because Git by default pipes all output through a pager so you see only one page of log output at a time.

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

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

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

You can also filter the list to commits that match some search criteria. The `--author` option allows you to filter on a specific author, and the `--grep` option lets you search for keywords in the commit messages. (Note that if you specify both author and grep options, the command will match commits with both.)

If you want to specify multiple grep options, you have to add `--all-match` or the command will match commits with either.

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

In Table 2-3 we'll list these and a few other common options for your reference.

```
Option  Description
-(n)    Show only the last n commits
--since, --after    Limit the commits to those whose CommitDate was made on-or-after the specifie
--until, --before   Limit the commits to those whose CommitDate was made on-or-before the specifi
--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.
```

2.3.2 Limiting Log Output according to Date/Time

To determine which commits in the Git source code repository (git://git.kernel.org/pub/scm/git/git.git) have CommitDate on 2014-04-29 relative to your local timezone (as set on your computer), use

```
$ git log --after="2014-04-29 00:00:00" --before="2014-04-29 23:59:59" \
  --pretty=fuller
```

As the output will be different according to the timezone where it will be run, it's recommended to always use an absolute time such as ISO 8601 format (which includes timezone information) as argument to `--after` and `--before`, so that everone running the command will get the same repeatable results.

To obtain commits made at a specific instant in time (e.g. 29 April 2013 at 17:07:22 CET), we can use

```
$ git log  --after="2013-04-29T17:07:22+0200"      \
          --before="2013-04-29T17:07:22+0200" --pretty=fuller

commit de7c201a10857e5d424dbd8db880a6f24ba250f9
Author:     Ramkumar Ramachandra <artagnon@gmail.com>
AuthorDate: Mon Apr 29 18:19:37 2013 +0530
Commit:     Junio C Hamano <gitster@pobox.com>
CommitDate: Mon Apr 29 08:07:22 2013 -0700

    git-completion.bash: lexical sorting for diff.statGraphWidth

    df44483a (diff --stat: add config option to limit graph width,
    2012-03-01) added the option diff.startGraphWidth to the list of
    configuration variables in git-completion.bash, but failed to notice
    that the list is sorted alphabetically.  Move it to its rightful place
    in the list.

    Signed-off-by: Ramkumar Ramachandra <artagnon@gmail.com>
    Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

The above times (`AuthorDate`, `CommitDate`) are displayed in default format (`--date=default`), which shows timezone information of respective author and commiter.

Other useful formats include `--date=iso` (ISO 8601), `--date=rfc` (RFC 2822), `--date=raw` (seconds since the epoch (1970-01-01 UTC)) `--date=local` (times according to your local timezone) as well as `--date=relative` (e.g. "2 hours ago").

When using `git log` without specifying time, the time defaults to the time at which the command is run on your computer (keeping the identical offset from UTC).

For example, running a `git log` at 09:00 on your computer with your timezone currently 3 hours ahead of UTC, makes the following two commands equivalent:

```
$ git log --after=2008-06-01 --before=2008-07-01
$ git log --after="2008-06-01T09:00:00+0300" \
    --before="2008-07-01T09:00:00+0300"
```

As a final example, if you want to see which commits modifying test files in the Git source code history were committed by Junio Hamano with CommitDate being in the month of October 2008 (relative to the timezone of New York) and were not merges, you can run something like this:

```
    $ git log --pretty="%h - %s" --author=gitster \
       --after="2008-10-01T00:00:00-0400"         \
      --before="2008-10-31T23:59:59-0400" --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 more than 36,000 commits in the Git source code history, this command shows the 6 that match those criteria.

2.3.3 Using a GUI to Visualize History

If you like to use a more graphical tool to visualize your commit history, you may want to take a look at a Tcl/Tk program called `gitk` that is distributed with Git. Gitk is basically a visual `git log` tool, and it accepts nearly all the filtering options that `git log` does. If you type `gitk` on the command line in your project, you should see something like Figure 2-2.

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 you the changes introduced at any commit you click.

2.4 Undoing Things

At any stage, you may want to undo something. Here, we'll review a few basic tools for undoing changes that you've made. Be careful, because you can't always revert some of these undos. This is one of the few areas in Git where you may lose some work if you do it wrong.

2.4.1 Changing Your Last Commit

One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to try that commit again, you can run commit with the `--amend` option:

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

This command takes your staging area and uses it for the commit. If you've made no changes since your last commit (for instance, you run this command

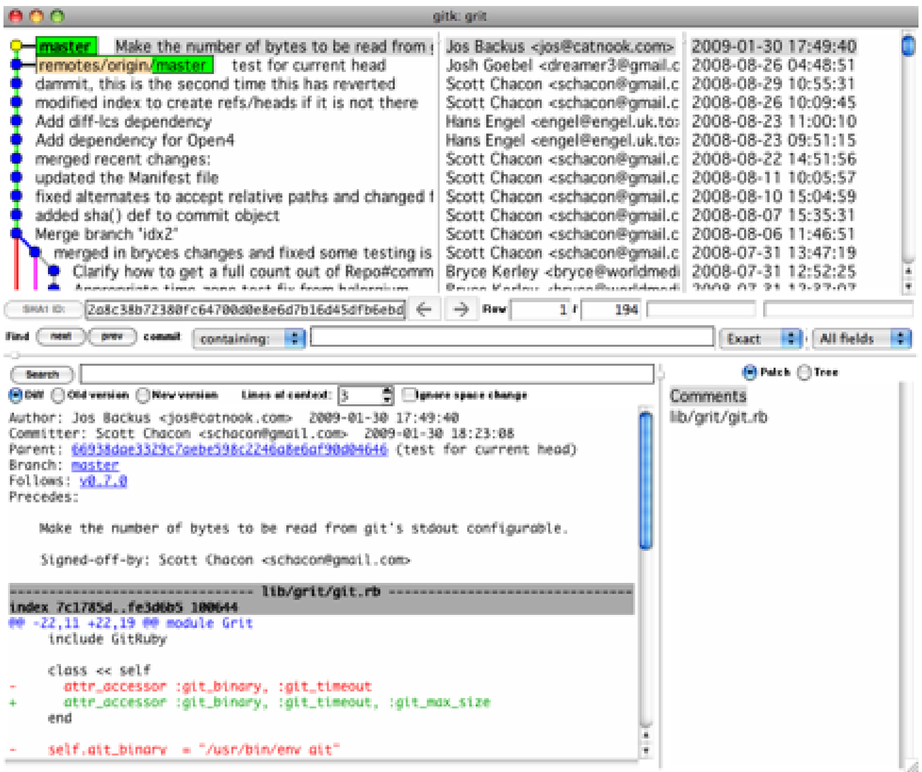

Figure 2.2: The gitk history visualizer.

immediately after your previous commit), then your snapshot will look exactly the same and all you'll change is your commit message.

The same commit-message editor fires up, but it already contains the message of your previous commit. You can edit the message the same as always, but it overwrites your previous commit.

As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

```
$ 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 results of the first.

2.4.2 Unstaging a Staged File

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

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

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

Right below the "Changes to be committed" text, it says "use `git reset HEAD <file>...` to unstage". So, let's use that advice to unstage the `benchmarks.rb` file:

```
$ git reset HEAD benchmarks.rb
Unstaged changes after reset:
M       benchmarks.rb
$ 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 command is a bit strange, but it works. The `benchmarks.rb` file is modified but once again unstaged.

2.4.3 Unmodifying a Modified File

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

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   benchmarks.rb
```

It tells you pretty explicitly how to discard the changes you've made (at least, the newer versions of Git, 1.6.1 and later, do this — if you have an older version, we highly recommend upgrading it to get some of these nicer usability features). Let's 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
```

You can see that the changes have been reverted. You should also realize that this is a dangerous command: any changes you made to that file are gone — you just copied another file over it. Don't ever use this command unless you absolutely know that you don't want the file. If you just need to get it out of the way, we'll go over stashing and branching in the next chapter; these are generally better ways to go.

Remember, anything that is committed in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an `--amend` commit can be recovered (see *Chapter 9* for data recovery). However, anything you lose that was never committed is likely never to be seen again.

2.5 Working with Remotes

To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Managing remote repositories includes 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, we'll cover these remote-management skills.

2.5.1 Showing Your Remotes

To see which remote servers you have configured, you can run the `git remote` command. It lists the shortnames of each remote handle you've specified. If you've cloned your repository, you should at least see *origin* — that is the default name Git gives to the server you cloned from:

```
$ git clone git://github.com/schacon/ticgit.git
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 193.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

You can also specify `-v`, which shows you the URL that Git has stored for the shortname to 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, the command lists them all. For example, my Grit repository looks something like this.

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

2.5.2 Adding Remote Repositories

I've mentioned and given some demonstrations of adding remote repositories in previous sections, but here is how to do it explicitly. To add a new remote Git repository as a shortname you can reference easily, 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 string `pb` on the command line in lieu of the whole URL. For example, if you want to fetch all the information that Paul has but that you don't yet have in your repository, you can 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
```

Paul's master branch is accessible locally as `pb/master` — you can merge it into one of your branches, or you can check out a local branch at that point if you want to inspect it.

2.5.3 Fetching and Pulling from Your Remotes

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

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

The command goes out to that remote project and pulls down all the data from that remote project that you don't have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time. (We'll go over what branches are and how to use them in much more detail in *Chapter 3*.)

If you clone a repository, the command automatically adds that remote repository under the name *origin*. So, `git fetch origin` fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It's important to note that the `fetch` command pulls the data to your local repository — it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready.

If you have a branch set up to track a remote branch (see the next section and *Chapter 3* for more information), you can use the `git pull` command to automatically fetch and then merge a remote branch into your current branch. This may be an easier or more comfortable workflow for you; and by default, the `git clone` command automatically sets up your local master branch to track the remote master branch on the server you cloned from (assuming the remote has a master branch). Running `git pull` generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you're currently working on.

2.5.4 Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: `git push [remote-name] [branch-name]`. If you want to push your master branch to your `origin` server (again, cloning generally sets up both of those names for you automatically), then you can run this to push your work back up to the server:

```
$ git push origin master
```

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to pull down their work first and incorporate it into yours before you'll be allowed to push. See *Chapter 3* for more detailed information on how to push to remote servers.

2.5.5 Inspecting a Remote

If you want to see more information about a particular remote, you can use the `git remote show [remote-name]` command. If you run this command with a particular shortname, such as `origin`, you get something like this:

```
$ 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
```

It lists the URL for the remote repository as well as the tracking branch information. The command helpfully tells you that if you're on the master branch and you run `git pull`, it will automatically merge in the master branch on the remote after it fetches all the remote references. It also lists all the remote references it has pulled down.

That is a simple example you're likely to encounter. When you're using Git more heavily, however, you may see much more information from `git remote show`:

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

This command shows which branch is automatically pushed when you run `git push` on certain branches. It also shows you which remote branches on the server you don't yet have, which remote branches you have that have been removed from the server, and multiple branches that are automatically merged when you run `git pull`.

2.5.6 Removing and Renaming Remotes

If you want to rename a reference, in newer versions of Git you can run `git remote rename` to change a remote's shortname. For instance, if you want to rename `pb` to `paul`, you can do so with `git remote rename`:

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

It's worth mentioning that this changes your remote branch names, too. What used to be referenced at `pb/master` is now at `paul/master`.

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

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

2.6 Tagging

Like most VCSs, Git has the ability to tag specific points in history as being important. Generally, people use this functionality to mark release points (`v1.0`, and so on). In this section, you'll learn how to list the available tags, how to create new tags, and what the different types of tags are.

2.6.1 Listing Your Tags

Listing the available tags in Git is straightforward. Just type `git tag`:

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

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

You can also search for tags with 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, you can run this:

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

```
v1.4.2.3
v1.4.2.4
```

2.6.2 Creating Tags

Git uses two main types of tags: lightweight and annotated. A lightweight tag is very much like a branch that doesn't change — it's just a pointer to a specific commit. Annotated tags, however, are stored as full objects in the Git database. They're checksummed; contain the tagger name, 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 have all this information; but if you want a temporary tag or for some reason don't want to keep the other information, lightweight tags are available too.

2.6.3 Annotated Tags

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

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

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

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

```
$ git show v1.4
tag v1.4
Tagger: 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 annotation message before showing the commit information.

2.6.4 Signed Tags

You can also sign your tags with GPG, assuming you have a private 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 can 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'
```

A bit later, you'll learn how to verify signed tags.

2.6.5 Lightweight Tags

Another way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file — no other information is kept. To create a lightweight tag, don't supply the `-a`, `-s`, or `-m` option:

```
$ git tag v1.4-lw
$ git tag
```

```
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

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

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

    Merge branch 'experiment'
```

2.6.6 Verifying Tags

To verify a signed tag, you use `git tag -v [tag-name]`. This command 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 get something like this instead:

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

2.6.7 Tagging Later

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

```
$ 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 tag the project at `v1.2`, which was at the "updated rakefile" commit. You can add it after the fact. To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

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

You can see that you've tagged the commit:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

2.6.8 Sharing Tags

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

```
$ git push origin v1.5
Counting objects: 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 that you want to push up at once, you can also use the `--tags` option to the `git push` command. This will transfer all of your tags to the remote server that are not already there.

```
$ git push origin --tags
Counting objects: 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 your repository, they will get all your tags as well.

2.7 Tips and Tricks

Before we finish this chapter on basic Git, a few little tips and tricks may make your Git experience a bit simpler, easier, or more familiar. Many people use Git without using any of these tips, and we won't refer to them or assume you've used them later in the book; but you should probably know how to do them.

2.7.1 Auto-Completion

If you use the Bash shell, Git comes with a nice auto-completion script you can enable. Download it directly from the Git source code at https://github.com/git/git/blob/master/contrib/completion/git-completion.bash . Copy this file to your home directory, and add this to your `.bashrc` file:

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

If you want to set up Git to automatically have Bash shell completion for all users, copy this script 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 will automatically load to provide shell completions.

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

Press the Tab key when you're writing a Git command, and it should return 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>` completes `git commit` automatically.

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

```
$ git log --s<tab><tab>
--shortstat               --sparse
--simplify-by-decoration  --src-prefix=
--simplify-merges         --stat
--since=                  --summary
```

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

2.7.2 Git Aliases

Git doesn't infer your command if you type it in partially. If you don't want to type the entire text of each of the Git commands, you can easily set up an alias for each command using `git config`. Here are a couple of examples you may want to set up:

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

This means that, for example, instead of typing `git commit`, you just need to type `git ci`. As you go on using Git, you'll probably use other commands frequently as well; in this case, don't hesitate to create new aliases.

This technique can also be very useful in creating commands that you think should exist. For example, to correct the usability problem you encountered with unstaging a file, you can add your own unstage alias to Git:

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

This makes the following two commands equivalent:

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

This seems a bit clearer. It's also common to add a `last` command, like this:

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

This way, you can see the last commit easily:

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

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

2.8 Summary

At this point, you can 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 the repository has been through. Next, we'll cover Git's killer feature: its branching model.

Chapter 3

Appendix A: Project Plan

The figure below shown the Gannt chart for our project plan.

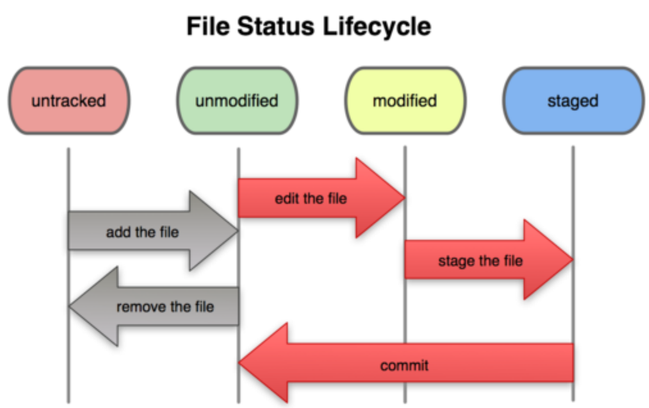

Figure 3.1: The lifecycle of the status of your files.

The main phases of the project were:

1. Requirements elicitation
2. Systems Analysis
3. Design
4. Implementation
5. Testing
6. Deployment

List of references

Adams, Douglas. 1979. *Hitch Hiker's Guide to the Galaxy.* UK: Pan.

Smith, Matt, and Chico Queiroz. 2015. *Unity 5.x Coobook.* UK: Packt.