Goal-Oriented Git

by George Brocklehurst

thoughtbot

Goal-Oriented Git

George Brocklehurst

January 15, 2016

Contents

About this book

If you want to learn how to use Git without worrying too much about what's going on under the hood, then this book is for you.

Every chapter will take something that Git can do for you—from tracking changes, to searching your files, to collaborating with others—and explain practically how to achieve that goal. We won't cover everything Git can do, but we won't just stick to the basics either: the goal of this book is to teach you a working set of Git commands that can get you through most day-to-day situations.

If you're learning Git for the first time, I'd recommend reading through in order: When we look at a particular goal, I'll assume knowledge of commands that were covered in earlier chapters.

Part I

Getting started

Goal: Get set up

What is Git?

Git is a version control system. When you work on a project—whether it's a piece of software, a Web site, a book, or anything else that goes through many revisions—using version control lets you track the changes that you make, return to previous versions, try alternative ideas, and seamlessly merge your own work with the work of others.

Git can track any kind of files, but it's best suited to plain text files. If you're not familiar with the term, these are files that just contain unformatted text as opposed to images or things like PDF.

Using the command line

Everything covered in this book will be done from the command line. While there are various applications that provide graphical interfaces for Git, the command line interface is powerful, flexible, and works on every platform. Issuing commands, one-by-one, forces us to take things step-by-step so we can really understand what's happening.

If you're not familiar with using command line tools, then there are a few terms you'll need to know. Here's a typical Git command:

```
git commit --message "My first commit"
```

We'll cover what it does later, but for now let's name its component parts:

- `git` is the command: it's the name of the program we want to run.
- `commit` is the sub-command: it tells Git what we want it to do. Not all command line interfaces use sub-commands, but since Git uses them for almost everything, in the context of this book we can just think of `git commit` as the command.
- `--message` and `"My first commit"` are both arguments: they are passed to the command to further nuance its behaviour. Arguments often come in pairs, where we tell Git what option we want to use—in this case `--message`—and then a value for that option—`"My first commit"`.

Installing and configuring Git

Before we begin, you'll need to install Git and get it set up:

1. Go to the Git downloads page, and follow the instructions for your operating system: http://git-scm.com/downloads

2. Make sure you can run Git commands from the command line. If you're on Mac OS X you can use the Terminal.app application; if you're on Windows there's an application called Git Bash that comes with Git.

   The following command should tell you what version of Git you have installed:

   ```
   $ git --version
   git version 2.2.1
   ```

   As with all of the examples in this book, the `$` at the beginning of the line indicates that this is a command; you shouldn't include it in the command you type.

   If you see a version number, as show above, then Git is installed successfully. If you see a message along the lines of `command not found`, then Git isn't correctly installed yet.

3. When Git is tracking changes to our files, it needs to know who made those changes—we'll see why this is important when we explore Git's collaborative features in Part 4.

   To identify yourself, and therefore the changes you track with Git, you need to set Git's `user.name` and `user.email` settings with the following commands, replacing my name and email address with your own:

   ```
   $ git config --global user.name "George Brocklehurst"
   $ git config --global user.email george@georgebrock.com
   ```

   The `--global` argument tells Git that these settings should apply to all of the Git projects you work on, so you'll only need to set this once. Don't worry if you don't completely follow this now; we'll talk about the `git config` command in more detail later in the book.

Goal: Track changes

If you can see how the incremental changes to your files led to their current state, then you have a better chance of understanding the project you are working on, fixing problems that arise, and collaborating successfully with others. In order to unlock the riches promised by a comprehensive history, we have to first build that history, one change at a time.

By now, you should have installed Git on your system, and so you are ready to create your first Git repository.

A repository is the place where Git keeps track of all of the changes for a given project. We need to create a repository for each project we want to track with Git.

The `git init` command

The `git init` command creates a new Git repository to keep track of changes to the files in the directory where the command is run. For example:

```
$ cd projects/git-book
$ git init
Initialized empty Git repository in /home/george/projects/git-book/.git/
```

This creates a new Git repository in the `projects/git-book` directory, telling Git that from now on we want to track the changes made to one or more of the files in that directory. Since we haven't told Git exactly which files to track yet, the repository is empty; this is true even if the `projects/git-book` directory isn't empty.

Running the command creates a new directory, `projects/git-book/.git`, where Git will store the files that represent the repository. This is the Git directory, whereas `projects/git-book` is the working directory.

To begin with, we'll modify files in the working directory, just as we would if we weren't using Git, and then tell Git to track the changes we've made in the repository. As we get more advanced, we'll be able to conjure old or alternative versions of our project from the repository, summoning them to the working directory to do our bidding.

The `git add` and `git commit` commands

The individual changes that make up the history of our project are called commits. We commit changes to the Git repository in the same way we might commit something to memory; once it's committed we won't lose it.

Each commit consists of a set of changes—like adding a new file, changing the contents of a file, deleting a file, or changing the properties of the file in some way—and a commit message that describes the changes, and the name and email address of the commit's author.

Git provides a staging area called the index, where we build up a set of changes before committing them to the repository. We'll come back to why the index is useful later, but for now it's enough to know that creating a commit is a three stage process:

1. Change files in the working directory.
2. Add some or all of those changes to the index.
3. Create a commit from the changes staged in the index.

Let's say we've created a file in our `git-book` working directory called `chapter1.txt`, and we're happy with the contents of the file and want to commit it to the repository. That's step one, step two is to add the file to the index using the `git add` command:

```
$ git add chapter1.txt
```

And then we can create a commit using the `git commit` command:

```
$ git commit --message "Add first draft of chapter one"
[master (root-commit) 6f9ec1c] Add first draft of chapter one
 1 file changed, 1 insertion(+)
 create mode 100644 chapter1.txt
```

The commit message, passed using the `--message` argument, describes the changes in the commit: in this case, it adds the first draft of chapter one.

By committing, we've drawn a line in the sand. Whatever changes we make to chapter one in future, we'll always be able to get back to this revision of the file. Even though a commit only contains the changes made to a file since it was last committed, we can think of a commit as representing a particular revision of our project, since Git is able to use all of the commits up to that point to reconstruct the revision of the project that existed when that commit was made.

Now that we've included it in a commit, the file `chapter1.txt` will be tracked by Git: if we make any changes to the file in future, Git will notice that it has changed and needs to be committed again.

Summary

- Create a new Git repository with `git init`
- Stage related changes in the index with `git add`
- Commit the changes in the index with `git commit`

Goal: Understand what is being tracked

So far, we've made simple commits with just one file, but in a complex project there could be dozens or even hundreds of files. How do we know if they've been changed since they were last committed, or if they're even tracked by Git?

In the last chapter we defined three important locations:

1. The working directory, where we keep and work on the current version of our project's files.
2. The index, where we build up a set of changes ready to commit.
3. The repository, where Git stores the history of our project as a series of commits.

Git provides a number of commands for comparing the state of the repository, the index, and the working directory, so we can stay up to date with what's changed.

The `git status` command

The `git status` command will tell us the status of our files. It lists all of the changed or untracked files in the working directory, split into the following groups:

1. **Changes to be committed**: files which have already been added to the index with `git add` and will be included the next time we run `git commit`.

2. **Changes not staged for commit**: files which have been changed in the working directory, but not added to the index.
3. **Untracked files**: files which have never been committed and therefore aren't tracked by Git.

For example:

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

    new file:   chapter2.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:   chapter1.txt

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

    chapter3.txt
```

In this case, `chapter2.txt` has just been added to the index for the first time with `git add`, `chapter1.txt` is tracked by Git but has been modified since it was last committed, and `chapter3.txt` isn't tracked by Git yet.

`git status` won't mention any files that haven't been changed since they were last committed. If none of your files have changed, `git status` will let you know that you have clean working directory:

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

`git status` isn't concerned with the details of exactly what's changed in each file, just which files have been changed and therefore could be committed.

The `git diff` command

Now we know which files in our working directory have changed, it would be good to see the details of those changes. The `git diff` command provides this.

If you've used the Unix `diff` utility then the output of `git diff` might look familiar, but while `diff` shows the difference between files, `git diff` shows the difference between versions of the same file.

Without any arguments, `git diff` shows all changes that haven't been committed or added to the index.

For example, here's what `chapter1.txt` the last time it was committed:

```
CHAPTER ONE

This is the first chapter, where it all beings.
```

Since that commit, we've modified `chapter1.txt` in the working directory to look like this:

```
CHAPTER 1

This is the first chapter, where it all beings.
```

Running `git diff` shows us exactly what the differences are:

```
$ git diff
diff --git a/chapter1.txt b/chapter1.txt
index 30c9d02..074d0db 100644
--- a/chapter1.txt
+++ b/chapter1.txt
@@ -1,3 +1,3 @@
-CHAPTER ONE
+CHAPTER 1

 This is the first chapter, where it all beings.
```

The first section gives some information about the change Git is showing: which versions are being compared, and on which lines of which files there are changes. Other than the file's name, this is more useful to machines than humans, and I usually don't read it carefully. More important are the changes themselves:

```
-CHAPTER ONE
+CHAPTER 1

 This is the first chapter, where it all beings.
```

Lines beginning with a `-` indicate a line that has been removed, and lines beginning with a `+` indicate a line that has been added. In this case a line has been changed: the old version of the line was removed, and the new version was added.

Lines beginning with a space—lines that lack both `+` and `-`—are unchanged, but are shown to give the context of the change.

The `git diff --staged` command

Remember that without any arguments `git diff` shows us the changes that haven't been staged in the index yet. `git diff --staged` shows us the changes that *have* been staged, but haven't yet been committed. In other words, `git diff --staged` shows you the changes that would be included in the commit, if you were to run `git commit` right now.

It can be useful to run `git diff --staged` before `git commit` as a final check that the right changes have been staged in the index.

Summary

- Look at the state of the working directory with `git status`
- Look at the changes since the latest commit with `git diff`
- Look at the contents of the index with `git diff --staged`

Goal: Make beautiful commits

A good commit should contain a set of related changes, with a description of why those changes were made. Much of the power of tracking history is lost if we bundle lots of unrelated changes together in the same commit, or don't take the time to describe them properly.

So far we've been assuming that all of the changes to a file belong in the same commit. Real world work is rarely this neat and tidy, though. Perhaps we were half way through changing a document, and found ourselves fixing an un-related typing error; perhaps we forgot to commit a set of changes before moving on to the next idea; perhaps we just got distracted: whatever the cause, a real project's working directory can be a messy place.

This is where the index really comes into its own: By updating the index before we commit, we can carefully select the changes that go into each commit; even going so far as to split unrelated changes to the same file over several commits.

The `git add --patch` command

In addition to the name of a file, the `git add` command can take some options to control exactly what is added to the index. Particularly useful is the `--patch` option, which shows you each change in turn and lets you decide if it should be added to the index or not.

Let's look at an example: I've made two changes to the same file but they are unrelated, so I want to add them to the repository as two separate commits. `git diff` tells me I have the following changes:

```
$ git diff
diff --git a/chapter1.txt b/chapter1.txt
index 8f6f18e..d119536 100644
--- a/chapter1.txt
+++ b/chapter1.txt
@@ -1,3 +1,3 @@
-The quick brown fxo jumped over the lazy dog.
+The quick brown fox jumped over the lazy dog.

-The dog was not best pleased.
+The dog was not best pleased, and barked angrily.
```

On the first line, I've corrected a typing error by changing `fxo` to `fox`, and then on the third line I've added some additional material about the dog's reaction to the fox. `git add --patch` will allow us to go through each change in turn, and decide if we want to add it to the index.

```
$ git add --patch
diff --git a/chapter1.txt b/chapter1.txt
index 8f6f18e..d119536 100644
--- a/chapter1.txt
+++ b/chapter1.txt
@@ -1,3 +1,3 @@
-The quick brown fxo jumped over the lazy dog.
+The quick brown fox jumped over the lazy dog.

-The dog was not best pleased.
+The dog was not best pleased, and barked angrily.
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

Git will show the changes—using the familiar style of `git diff`—one at a time, and asks me what I want to do with each hunk with the question `Stage this hunk [y,n,q,a,d,/,s,e,?]?`. Most of the time we can answer with `y` to add the change to the index, or `n` to ignore it for now and move on to the next change. In this case, it's a little more complicated: since my changes are very close together, Git's assumed that they're probably related and shows them as a single hunk. I want to split up this hunk, and since I need something beyond a basic "yes" or "no" I can use `?` to explain the other options:

```
Stage this hunk [y,n,q,a,d,/,s,e,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

After looking through my options, I see I can use `s` to tell Git that even though these changes are close to each other they should be split up so that I can treat them differently:

```
Stage this hunk [y,n,q,a,d,/,s,e,?]? s
Split into 2 hunks.
@@ -1,2 +1,2 @@
-The quick brown fxo jumped over the lazy dog.
+The quick brown fox jumped over the lazy dog.

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

Git splits the change into two hunks, and shows me the first of them. This change is just the correction on the first line, so I can stage it in the index using `y`. Once it's staged, Git will show me the next change. Since this change isn't related to the one I've already added to the index, I don't want to include it in the same commit, so I can skip it with `n`:

```
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? y
@@ -2,2 +2,2 @@
```

```
-The dog was not best pleased.
+The dog was not best pleased, and barked angrily.
Stage this hunk [y,n,q,a,d,/,K,g,e,?]? n
```

I've now made a decision about each of the changes to the tracked files in my working directory, so `git add --patch` exits, and I can commit the correction that's staged, and then stage and commit the additional content:

```
$ git commit --message "Fix typing error"
[master ee02d29] Fix typing error
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git diff
diff --git a/chapter1.txt b/chapter1.txt
index 9d07dc3..d119536 100644
--- a/chapter1.txt
+++ b/chapter1.txt
@@ -1,3 +1,3 @@
 The quick brown fox jumped over the lazy dog.

-The dog was not best pleased.
+The dog was not best pleased, and barked angrily.
$ git add chapter1.txt
$ git commit --message "Add information about the dog's reaction"
[master 1caf774] Add information about the dog's reaction
 1 file changed, 1 insertion(+), 1 deletion(-)
```

I almost always use the `--patch` option when I'm adding files to the index, even when I'm confident that all of the changes are related and will end up as a single commit: It gives me the opportunity to review my changes, and has saved me many times from committing something that wasn't quite right.

Summary

- Good commits are focused on a set of related changes
- Use `git commit --patch` to add specific changes to the index

Goal: Write beautiful commit messages

In the last chapter we talked about the importance of carefully choosing the changes that go into each commit: we wanted each change to be easy to understand in isolation so our history would be easier to work with.

Carefully selecting the contents of the commit is only half the story: writing descriptive commit messages is also important for building an understandable—and therefore useful—history.

What to cover

When we look back at commits, Git can show us *what* has changed by showing us the contents of the commit, but it cannot show us *why* unless we use our commit messages to explain the reasoning behind each change.

You should write commit messages that address the following questions:

- What need do the changes in this commit address?
- If it's not clear from looking at the changes in the commit, how do the changes address that need?
- What other consequences might this change have?

That list of questions might seem like a lot, but don't be afraid to write a long commit message if you need to. The next time you're trying to remember why you

changed your project in a certain way you'll be very glad you took the time to write it down.

Thinking through your change as you commit it to Git can also be a useful opportunity to review your work, and catch mistakes or consequences you didn't think of earlier.

How to lay it out

The first line of a commit message should be a short summary of the commit's intention, no more than 50 characters long; think of it like the subject line of an email. When we start looking back at the commits we've made—we'll see how in the next section, "Using history"—Git will often only show the first line of the commit message, so making it short and meaningful will keep those lists of commits tidy and useful.

After the one-line summary and a blank line, we can write a longer description of the commit, often running to several paragraphs or lists of bullet points; think of this like the body of an email. It may be relevant to include Web links to additional information here. In a software project a problem has often been discussed on a bug tracking Web site before it is fixed, and including a link to this discussion when committing the changes that fix the problem can be very valuable.

It's customary to keep each line of the description to 72 characters or fewer: as with the one line summary, this length limit will keep the body of the commit message legible in all the various contexts where it might appear.

As a concrete example, when I commit this chapter to Git, the message could look something like this:

```
First draft of "Write beautiful commit messages"

Vague commit messages aren't very useful. This chapter attempts to
impress upon readers the importance of detailed and specific commit
messages.

* Add chapter 1.5 (beautiful commit messages).
* Update chapter 2.1 (read the history) to refer back to chapter 1.5.
```

Write commit messages in your text editor

So far, we've used the `--message` argument to specify a simple, one-line commit message when we make a commit.

If you omit this argument, Git will open your text editor and prompt you to write a commit message there. Writing the commit message in an editor is almost always preferable to using `--message`: it gives you the time and space to think through the changes you've made, and describe them in more than a single short line.

Many text editors also have settings that help enforce the line-length conventions discussed above, so you can focus on what you're writing, and let your editor worry about whether or not it's time to insert a new line.

When you've finished writing the commit message, just save the file and close your editor. Behind the scenes, Git has given your editor a temporary file where you can write your message, and when you save and close that file, Git will extract your message from it.

Set your text editor

If you're on a Linux or Unix system, and you frequently use the command line, chances are you've already set an `$EDITOR` or `$VISUAL` environment variable. Git will pick up on that setting and use it.

If you're on a different operating system, unfamiliar with setting environment variables, or you want to use a different editor for Git commit messages, you can explicitly tell Git which editor to use. This is done by setting `core.editor` with the `git config` command:

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

The `--global` argument tells Git that this setting should apply to all of your repositories, without it Git would change this setting only for the current repository.

`core.editor` is the name of the setting, and in this case I'm telling Git that the command to open my editor is `vim`. You should replace `vim` with whatever command you use to launch your favourite text editor from the command line. For example if you use the Sublime Text editor you might use:

```
$ git config --global core.editor "subl -n -w"
```

The `git commit --verbose` command

In order to write such a detailed commit message, it can be useful to refer to the changes you are committing while you write the commit message. If we pass the `--verbose` argument to `git commit`, then it will include a list of the changes we are committing—again, in that familiar `git diff` style—at the bottom of the file Git opens in our editor.

When using the `--verbose` option, Git won't include the list of changes, or anything you add after it, in the commit message. Make sure you write your commit message at the top of the file.

Git will helpfully mark the place where the message ends and the summary of the changes starts with an ASCII art pair of scissors cutting along a dotted line:

```
# ------------------------ >8 ------------------------
# Do not touch the line above.
# Everything below will be removed.
```

Summary

- Write commit messages that describe why changes were made.
- The first line of the commit message should be a short summary, 50 characters or fewer.
- Use `git config --global core.editor <editor>` to set your text editor.
- Use `git commit --verbose` to see the changes you are committing in your editor.

Part II

Using history

Goal: Read the history

Now that we've made some commits, we can begin to use them to explore the history of our project.

You'll often want to view your whole commit history. It can be a great way of refreshing your memory about what you were working on, or seeing what others have done on a collaborative project.

The `git log` command

The `git log` command will show all of the commits, with their authors, dates, and messages:

```
$ git log
commit 2a40e706eaf92805907a1f3be707a199982ef0f9
Author: George Brocklehurst <george@georgebrock.com>
Date:   Mon Oct 27 15:18:26 2014 -0400

    Draft content for chapter 2

commit c7d5d6876da96f8aff9b08416119ff0178d776cf
Author: George Brocklehurst <george@georgebrock.com>
Date:   Mon Oct 27 15:16:48 2014 -0400

    Draft content for chapter 1
```

In this example, the project has two commits. The most recent commit is shown first.

Each commit starts with an long string of numbers and letters that uniquely identifies the commit:

```
commit 2a40e706eaf92805907a1f3be707a199982ef0f9
```

These identifiers are very important. Many of the ways we can use our projects' Git history involve referring to specific commits, and we often do that using these unique identifiers.

The `--oneline` option

In the last chapter we discussed detailed commit messages, which can often get very long. If every commit message runs to dozens of lines, the full output of `git log` can be a little overwhelming, and difficult to quickly scan for a particular commit.

The `git log` command accepts many options to customise the format of the output; one of the most useful is `--oneline`, which only outputs the first line of the commit message, and the first few characters of the unique identifier:

```
$ git log --oneline
2a40e70 Draft content for chapter 2
c7d5d68 Draft content for chapter 1
```

Remember that the first line of the commit message should be a short summary of the commit, so this is a great way to get an overview of recent changes.

Running `git help log` will give you comprehensive documentation on the other ways you can customise the output of `git log`.

The `git show` command

While `git log` gives us an overview of many commits, the `git show` command will show the contents of a specific commit. We need to tell it which commit to show;

this is the first of many examples of a place where we can use a commit's unique identifier.

```
$ git show c7d5d6876da96f8aff9b08416119ff0178d776cf
commit c7d5d6876da96f8aff9b08416119ff0178d776cf
Author: George Brocklehurst <george@georgebrock.com>
Date:   Mon Oct 27 15:16:48 2014 -0400

    Draft content for chapter 1

diff --git a/chapter1.txt b/chapter1.txt
new file mode 100644
index 0000000..b5689b5
--- /dev/null
+++ b/chapter1.txt
@@ -0,0 +1 @@
+Chapter 1: This is the first chapter.
```

The output should all be familiar: it starts with information about the commit, which is very similar to the output produced by `git log`, and then shows the changes made by the commit, which is very similar to the output produced by `git diff`.

Summary

- Use `git log` to see a list of commits.
- Use `git log --oneline` to see just the first line of the commit messages.
- Use `git show <identifier>` to see the contents of a specific commit.

Goal: Refer to commits

In the last chapter we learnt how to use the `git log` command to find a commit's identifier, which we could use with the `git show` command to view the contents of that commit.

Commit identifiers can be quite cumbersome to use: they're long, complex, and not very memorable. Fortunately, Git provides a range of alternatives.

Let's imagine the following Git history, and see how we can refer to the commits without using the long-winded commit identifiers:

```
$ git log
commit e420911b9d16d0fd75a12a71202673ae32fc933a
Author: George Brocklehurst <george@georgebrock.com>
Date:   Mon Oct 27 15:25:32 2014 -0400

    Third commit

commit 25cece8870c50144c68b5fe27a03aeb647b28c4a
Author: George Brocklehurst <george@georgebrock.com>
Date:   Mon Oct 27 15:25:29 2014 -0400

    Second commit

commit d8084d3fb7f9924dcefad1a20da885b1aba28d54
Author: George Brocklehurst <george@georgebrock.com>
Date:   Mon Oct 27 15:25:23 2014 -0400
```

```
    First commit
```

Abbreviating commit identifiers

Let's say we want to look at the latest commit. What we've seen so far is `git show` with the full identifier:

```
$ git show e420911b9d16d0fd75a12a71202673ae32fc933a
commit e420911b9d16d0fd75a12a71202673ae32fc933a
Author: George Brocklehurst <george@georgebrock.com>
Date:   Mon Oct 27 15:25:32 2014 -0400

    Third commit
```

In the previous chapter we saw that the `git log --oneline` command only prints the start of the commit identifier. It does this because it's possible to use the beginning of the identifier to identify a commit, as long as you use enough of it to be unique among all of the commit identifiers in the repository. In practice, this usually means that we can use the first 6 or 7 characters of the identifier without any problems, even in a long running project with thousands of commits.

```
$ git show e420911
commit e420911b9d16d0fd75a12a71202673ae32fc933a
Author: George Brocklehurst <george@georgebrock.com>
Date:   Mon Oct 27 15:25:32 2014 -0400

    Third commit
```

If you don't give enough characters to uniquely identify a single commit, Git will let you know that you've provided an ambiguous argument:

```
$ git show e42
fatal: ambiguous argument 'e42': unknown revision or path not in the working tree.
Use '--' to separate paths from revisions, like this:
'git <command> [<revision>...] -- [<file>...]'
```

Relative commit references

Each commit in our Git repository has a parent commit: the commit that immediately preceded it. If we know the identifier of a commit, we can tell Git to give us its parent commit using a `~1` suffix. For example:

```
$ git show e420911~1
commit 25cece8870c50144c68b5fe27a03aeb647b28c4a
Author: George Brocklehurst <george@georgebrock.com>
Date:   Mon Oct 27 15:25:29 2014 -0400

    Second commit
```

By increasing the number after the `~` we can follow the parent relationships back through the project's history. For example, `~2` will give us the parent of the parent commit:

```
$ git show e420911~2
commit d8084d3fb7f9924dcefad1a20da885b1aba28d54
Author: George Brocklehurst <george@georgebrock.com>
Date:   Mon Oct 27 15:25:23 2014 -0400

    First commit
```

Using the commit message

These techniques are all great if you know the commit's identifier, or at least the identifier of one of its descendants, but identifiers aren't as memorable as commit messages. Fortunately it's possible to refer to commits using a part of their message, too:

```
$ git show :/Second
commit 25cece8870c50144c68b5fe27a03aeb647b28c4a
Author: George Brocklehurst <george@georgebrock.com>
Date:   Mon Oct 27 15:25:29 2014 -0400
```

```
    Second commit
```

If there are multiple commits that match the given word, then Git will pick the newest matching commit.

There are more!

We've only scratched the surface of the many ways of referring to Git commits. If you want all the options, then Git's comprehensive help system is the place to go:

```
$ git help revisions
```

If you're reading this book in order to learn Git for the first time, then many of the concepts in that document won't be familiar to you yet, but don't be afraid to explore the documentation.

Summary

- Use abbreviated commit identifiers.
- Use relative commit references, e.g. `e420911~2`.
- Refer to commits using their message, e.g. `:/Second`.
- Check `git help revisions` for more.

Goal: Understand a change

Now that we've learnt how to look at the whole history, and a variety of ways to refer to a specific commit, we're ready to start asking more demanding questions of our Git repository.

It can often be the case in a long running project that we come across something in one of our files that doesn't quite make sense. Using the project's Git history to see where something came from can help to make it more clear. Git can help us answer questions like "why was this changed?" and "what was here before it changed?"

Here's a confusing file called `chapter1.txt`. Let's dig into the history and see if we can work out what's going on:

```
There are several things you should know:
Lastly, Git will help you get your work done.
```

It looks like there should be a list here, but there's only one item, and confusingly it seems like it should be the last item. Maybe this just needs to be rephrased, but maybe something was deleted that shouldn't have been.

The `git blame` command

The first command we can use to understand this file is `git blame`, which will tell us which person, and more importantly which commit, is to blame for the current state of each line in the file:

```
$ git blame chapter1.txt
^61966bf (George Brocklehurst 2014-10-23 11:21:55 -0400 1) There are several things you shou
^61966bf (George Brocklehurst 2014-10-23 11:21:55 -0400 2) Lastly, Git will help you get you
```

Each line of the file is printed with some annotations: the abbreviated identifier of the commit in which it was last changed, the author and date of that commit, and the line number.

In this case, both of those lines were last changed in the commit `61966bf`. The next step in our investigation should probably be to look at that commit and see the full change:

```
$ git show 61966bf
commit 61966bf14e754946d0ab3a6d8e66c419918c15a8
Author: George Brocklehurst <george@georgebrock.com>
Date:   Thu Oct 23 11:21:55 2014 -0400

    List things readers should know about Git.

diff --git a/chapter1.txt b/chapter1.txt
new file mode 100644
index 0000000..ecefc9c
--- /dev/null
+++ b/chapter1.txt
@@ -0,0 +1,4 @@
+There are several things you should know:
+Firstly, Git is not very good.
+Secondly, Git is quite useful.
+Lastly, Git will help you get your work done.
```

This is a clue on the road to solving this mystery: we now know that the ambiguous section started as a list, but a couple of items were subsequently removed. It's a good start, but we still don't know why the list was altered. Maybe the correct fix is to rephrase the content, but maybe those list items should never have been removed in the first place.

To find out where the list items went, we can turn to another command we've used before: `git log`.

Filtering `git log` by commit and file name

We've used `git log` to look at all of the commits in our project. In this case, we want something more specific. The `git log` command accepts a range of commits and a file path as arguments, which we can use to only see commits that:

1. happened *after* the list was first introduced in commit `61966bf`, and
2. contain changes to the file `chapter1.txt`.

The most common way to refer to a range of commits is `first..last`, where `first` and `last` both refer to commits, using their identifiers, or one of the other methods we've seen of identifying a commit. Git also allows us to omit the end of the range, assuming when we do so that we mean it to end with the most recent commit. In our case, we want to see everything after commit `61966bf`, which introduced the list, so we can use the range `61966bf..`.

Putting this together with the file path `chapter1.txt`, we get this command:

```
$ git log 61966bf.. chapter1.txt
commit aaad9ad5001689eeaeca0ef88626debd27801aea
Author: George Brocklehurst <george@georgebrock.com>
Date:   Thu Oct 23 11:22:31 2014 -0400

    Remove contradictory information.
```

Our carefully filtered log tells us that only one commit touched the file we're interested in during the time we're interested in. Let's take a closer look:

```
$ git show aaad9ad
commit aaad9ad5001689eeaeca0ef88626debd27801aea
Author: George Brocklehurst <george@georgebrock.com>
Date:   Thu Oct 23 11:22:31 2014 -0400

    Remove contradictory information.

diff --git a/chapter1.txt b/chapter1.txt
```

```
index ecefc9c..60ab488 100644
--- a/chapter1.txt
+++ b/chapter1.txt
@@ -1,4 +1,2 @@
 There are several things you should know:
-Firstly, Git is not very good.
-Secondly, Git is quite useful.
 Lastly, Git will help you get your work done.
```

Finally we have the full context of this confusing file. It started out as a list, but a couple of items were removed because they were contradictory, which left the surrounding content in a confusing state. Now that we have the full story, we can confidently fix the problem.

Not every investigation will follow these steps, but if you understand the commands we've used here you should be able to track down the source of all kinds of problems.

Summary

- Use `git blame <file>` to find out which commit last changed the lines in a file.
- Use `git log <range> <file>` to see relevant commits.

Goal: Search the repository

In the last chapter, we saw how to filter the output of `git log` to only show us commits in a certain range, or commits that contain changes to certain files, so we could see what changes had been made. In this chapter we'll start from the other side: we know the change we're interested in, but we don't know when or where it happened.

Git provides some powerful search tools which let us search through the files it is tracking and the commits we've made in the past.

Filtering `git log` by content

Let's pretend I'm writing a novel, which has the following Git history:

```
$ git log --oneline
2ba40c1 Remove cafe scene and references
e8393d4 Refer back to the cafe in chapter 2.
1dba13c Add cafe scene
3ba3f98 Remove zoo scene
0e6d7f7 Begin work on chapter 2
597b543 Begin work on chapter 1
```

I deleted a section of the novel, but I'm having second thoughts: I think the story will flow better if I put it back. Unfortunately, I deleted it some time ago, and I don't remember all of the details, just that it centred around a character called Alice.

There are a few commit messages that mention removing content, but by filtering the commit log to only show commits which contain changes that add or remove the word "Alice", I can narrow down my search. We do this using `git log`'s `-S` option:

```
$ git log --oneline -S Alice
3ba3f98 Remove zoo scene
0e6d7f7 Begin work on chapter 2
597b543 Begin work on chapter 1
```

Now I've narrowed down the search to three commits, and only one of the messages mentions removing content, so it looks like I've found my deleted scene.

```
$ git show 3ba3f98
commit 3ba3f9832331a486d8aeabe6215b3abaa1dfc052
Author: George Brocklehurst <george@georgebrock.com>
Date:   Fri Oct 31 16:44:34 2014 -0400

    Remove zoo scene

diff --git a/chapter2.txt b/chapter2.txt
index 2ba6e57..4961d74 100644
--- a/chapter2.txt
+++ b/chapter2.txt
@@ -1,4 +1,3 @@
 CHAPTER TWO

-Alice and Bob went to the zoo. Alice enjoyed learning about the different
-animals, while Bob mostly just enjoyed eating icecream.
+We all know that Bob likes icecream.
```

The `-S` option can be combined with the other filtering techniques we've seen, so if I had remembered the file that contained the change, or the range of commits it was part of, I could have narrowed down the search even further.

You can think of filtering the `git log` as searching in time: Given its knowledge of the history of your project, you're asking Git to tell you about times when certain

kinds of changes were made. It's also useful to be able to search in space: given a fixed point in time—in Git's terminology, a specific revision of the project—where does a certain word or phrase appear in the project's tracked files?

The `git grep` command

The `git grep` command searches through all of the tracked files for a particular word or phrase. If you're familiar with the Unix `grep` utility, you'll find that `git grep` works in much the same way.

Let's say we want to find all references to Alice in all of the tracked files in the working directory. We can use `git grep`:

```
$ git grep Alice
chapter1.txt:There were once two people called Alice and Bob.
```

This tells me that Alice is currently only mentioned once, in the file `chapter1.txt`. I know that the commit `3ba3f98` removed some mentions of Alice—we just used the `git log -S` command to find it—and I want to find out how often she was mentioned before that commit.

If you've read the chapter on how to "Refer to commits", you'll know that if we have the ID of a commit, we can use a `~1` suffix to reference the commit before that commit. In this case, we're interested in the revision before `3ba3f98`, so we can use `3ba3f98~1`.

The `git grep` command can take a commit as a second argument, which tells it to search the particular revision of our project represented by that commit:

```
$ git grep Alice 3ba3f98~1
3ba3f98~1:chapter1.txt:There were once two people called Alice and Bob.
3ba3f98~1:chapter2.txt:Alice and Bob went to the zoo. Alice enjoyed learning about the diffe
```

I've found my answer: Alice used to be mentioned in `chapter1.txt` and `chapter2.txt`.

`git grep` can match more than just simple words and phrases: it supports a powerful pattern matching language called Regular Expressions, which means you can

search for things like "any two digit number", or "either the word 'Alice' or the word 'Bob'". How to write regular expressions is beyond the scope of this book, but there is plenty of information on the Web, and even whole books on the topic.

Summary

- Find commits that mention a word or phrase with `git log -S <word>`.
- Find tracked files that mention a word or phrase with `git grep <word>`.
- Search previous revisions with `git grep <word> <revision>`.

Goal: Undo changes

In the previous chapter, we saw how to use Git's search tools to find a change we were having second thoughts about. Having found the commit, how can we reverse that change? We could use `git show` to view the changes it introduced and manually unpick them by typing them out again or using copy-and-paste. That would be a lot of manual, error-prone work, and fortunately Git provides tools to undo changes we'd rather hadn't been committed.

The `git revert` command

The simplest way of undoing the changes in a commit is to use the `git revert` command. This command creates a new commit which contains the opposite set of changes to the commit we want to undo: if a line was added in the original commit, it will be removed in the new commit, and vice versa.

Here is the commit we identified in the previous chapter:

```
$ git show 3ba3f98
commit 3ba3f9832331a486d8aeabe6215b3abaa1dfc052
Author: George Brocklehurst <george@georgebrock.com>
Date:   Fri Oct 31 16:44:34 2014 -0400

    Remove zoo scene

diff --git a/chapter2.txt b/chapter2.txt
```

```
index 2ba6e57..4961d74 100644
--- a/chapter2.txt
+++ b/chapter2.txt
@@ -1,4 +1,3 @@
 CHAPTER TWO

-Alice and Bob went to the zoo. Alice enjoyed learning about the different
-animals, while Bob mostly just enjoyed eating icecream.
+We all know that Bob likes icecream.
```

When we pass the commit's identifier as an argument to the `git revert` command, Git will construct the opposite set of changes, and commit them.

```
$ git revert 3ba3f98
[master 786b2c3] Revert "Remove zoo scene"
 1 file changed, 2 insertions(+), 1 deletion(-)
```

Notice that the output of the `git revert` command contains the identifier of the new commit.

```
$ git show 786b2c3
commit 786b2c3fa72af8372729a2e789a0071bba92de6b
Author: George Brocklehurst <george@georgebrock.com>
Date:   Fri Jan 15 13:50:38 2016 -0500

    Revert "Remove zoo scene"

    This reverts commit 3ba3f9832331a486d8aeabe6215b3abaa1dfc052.

diff --git a/chapter2.txt b/chapter2.txt
index 4961d74..2ba6e57 100644
--- a/chapter2.txt
+++ b/chapter2.txt
@@ -1,3 +1,4 @@
 CHAPTER TWO

-We all know that Bob likes icecream.
```

```
+Alice and Bob went to the zoo. Alice enjoyed learning about the different
+animals, while Bob mostly just enjoyed eating icecream.
```

Like the `git commit` command, `git revert` will open a text editor so that we can write a description of the commit, but unlike `git commit` it helpfully provides a default commit message with some information about the commit that is being reverted. It's usually a good idea to leave the default message intact—so that you'll be able to recognise a revert commit when you're reading the history—and to add a note to the end to explain *why* it was useful to revert this commit.

The `git reset` command

While `git revert` is useful for undoing a commit from somewhere back in the distant history of the project, we have other options if we want to undo our most recent commit or commits.

The `git reset` command will reset the repository back to an earlier state. It's a flexible command that can be used in various ways; we'll see it again later in the book, but for now let's focus on two uses:

1. To remove changes from the index: they will still exist in the working directory, but they won't be included in the next commit.
2. To remove the last commit or commits from the repository: the commit's changes will still exist in the working directory, but it will be as if they were never committed.

Remove changes from the index

Without any arguments, `git reset` will clear the index without changing the working directory or repository: it undoes the `git add` command. This kind of reset is useful when you're staging some changes, but you realise that you're not really ready to commit them yet, or that you've staged the wrong changes.

Continuing with my novel, I've made some changes to `chapter1.txt` that I think are ready to commit, so I add them to the index, and `git status` shows that they are staged to commit:

```
$ git add chapter1.txt
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   chapter1.txt
```

I was about to commit, but I've changed my mind: these changes aren't all related, and it would probably be better to split them up over multiple commits. I can remove them from the index using `git reset`:

```
$ git reset
Unstaged changes after reset:
M   chapter1.txt
```

`git status` confirms that `chapter1.txt` has still been changed, but is no longer staged to commit:

```
$ 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:   chapter1.txt

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

Remove commits from the repository

When it's given a commit as an argument, `git reset` will clear the index, and roll the repository back to that revision, as if the subsequent commits had never happened. It won't change the working directory, so the current state of our files won't be lost, but the history of how we got to that state will be.

This kind of reset is useful when you realise that your last commit was too big, and you'd like to break it down as several smaller commits. Or maybe you realise that the changes in your last two commits would make more sense spread across three commits.

Although undoing commits is useful, it should make us a little uncomfortable. Commits are the record of how our project has changed, and if we undo them there's a very real chance that we'll lose some valuable history along the way. Over the last few chapters we've seen a few of the ways that having access to a project's history can help us; hopefully we've seen enough that the idea of throwing part of the history away gives us pause.

Let's see how this works with a real example: I want to undo the last two commits to my novel. First, to refresh our memory, let's look at what those commits contain:

```
$ git log --oneline
2ba40c1 Remove cafe scene and references
e8393d4 Refer back to the cafe in chapter 2.
1dba13c Add cafe scene
3ba3f98 Remove zoo scene
0e6d7f7 Begin work on chapter 2
597b543 Begin work on chapter 1
```

Remember that `git log` shows the most recent commit first, so I'm trying to undo `e8393d4` and `2ba40c1`. The first of these commits adds some information about what Bob does at the cafe:

```
$ git show e8393d4
commit e8393d449377539aeda017d8fbf2fcefe76b9b4c
Author: George Brocklehurst <george@georgebrock.com>
Date:   Fri Oct 31 16:52:30 2014 -0400

    Refer back to the cafe in chapter 2.

diff --git a/chapter2.txt b/chapter2.txt
index 4961d74..07d4e18 100644
--- a/chapter2.txt
+++ b/chapter2.txt
```

```
@@ -1,3 +1,4 @@
 CHAPTER TWO

-We all know that Bob likes icecream.
+We all know that Bob likes icecream. Sometimes, Bob likes to eat icecream at the
+cafe.
```

The second commit removes all references to the cafe:

```
$ git show 2ba40c1
commit 2ba40c104744d3f748ea67ad5c75f16dc7840d78
Author: George Brocklehurst <george@georgebrock.com>
Date:   Fri Oct 31 16:53:19 2014 -0400

    Remove cafe scene and references

diff --git a/chapter1.txt b/chapter1.txt
index aa87c61..0ff7cf0 100644
--- a/chapter1.txt
+++ b/chapter1.txt
@@ -1,6 +1,3 @@
 CHAPTER ONE

 There were once two people called Alice and Bob.
-
-There is a cafe in the town where they live, and Bob likes to go there on
-weekends.
diff --git a/chapter2.txt b/chapter2.txt
index 07d4e18..4961d74 100644
--- a/chapter2.txt
+++ b/chapter2.txt
@@ -1,4 +1,3 @@
 CHAPTER TWO

-We all know that Bob likes icecream. Sometimes, Bob likes to eat icecream at the
-cafe.
+We all know that Bob likes icecream.
```

In order to undo these commits, I need to reset the repository to the last revision before they were made: `1dba13c`.

```
$ git reset 1dba13c
Unstaged changes after reset:
M   chapter1.txt
```

This doesn't change the working directory, but I have lost some information in the process. Let's look at the state of the repository after I reset:

```
$ git log --oneline
1dba13c Add cafe scene
3ba3f98 Remove zoo scene
0e6d7f7 Begin work on chapter 2
597b543 Begin work on chapter 1
```

The log looks good: the two commits we wanted to undo are gone. How about the working directory?

```
$ 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:   chapter1.txt

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

The status looks OK, too: there are some uncommitted changes in the working directory, but the commits we just removed made changes to both `chapter1.txt` and `chapter2.txt`, and yet our working directory only has changes to `chapter1.txt`:

```
$ git diff
diff --git a/chapter1.txt b/chapter1.txt
index aa87c61..0ff7cf0 100644
```

```
--- a/chapter1.txt
+++ b/chapter1.txt
@@ -1,6 +1,3 @@
 CHAPTER ONE

 There were once two people called Alice and Bob.
-
-There is a cafe in the town where they live, and Bob likes to go there on
-weekends.
```

The changes to `chapter2.txt` in our two deleted commits cancelled each other out: the first commit added a sentence about what Bob did at the cafe, and the second took it away again. By using `git reset` to remove these two commits, we've lost a potentially useful bit of history; there's no longer any record that this sentence ever existed, which means we no longer have the option of recovering it if we change our minds. Equally importantly, the information associated with the two commits is gone too: we no longer know who made those commits, when, or why.

`git reset` is a useful tool, but it should be used with caution, and is least risky when used to undo a single commit immediately after it was made. Git is a powerful tool, and will often give us more than enough rope to hang ourselves.

Part III

Branching

Introduction

STUB

Goal: Create a branch

STUB

Goal: Compare branches

STUB

Goal: Combine branches

STUB

Goal: Move branches

STUB

Goal: Delete branches

STUB

Part IV

Collaboration

Introduction

STUB

Goal: Add a remote

STUB

Goal: Publish changes

STUB

Goal: Retrieve changes

STUB