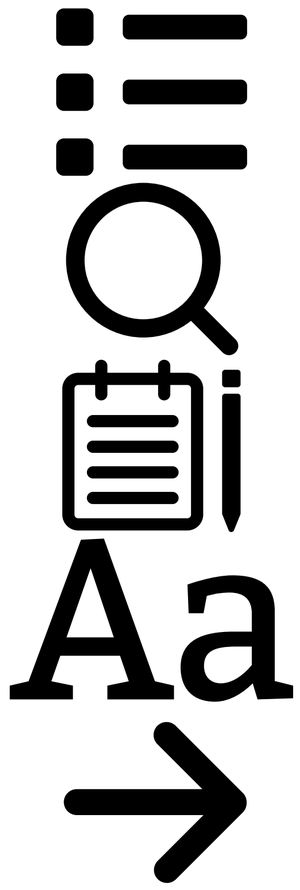

6
Git Log & History Written by Bhagat Singh & Chris Belanger

You've been quite busy in your repository, adding files, making changes, undoing changes and making intelligent commits with good, clear messages. But as time goes on, it gets harder and harder to remember what you did and when you did it.
When you mess up your project (not if, but *when*), you'll want to be able to go back in history and find a commit that worked, and rewind your project back to that point in time. This chapter shows you how.

Viewing Git history

Git keeps track of pretty much everything you do in your repository. You've already seen this in action in previous chapters, when you used the `git log` command.
However, there are many ways you can view the data provided by `git log` that can tell you some incredibly interesting things about your repository and your history. In fact, you can even use `git log` to create a graphical representation of your repository to get a better mental image of what's going on.

Vanilla git log

Open your terminal app and execute `git log` to see the basic, vanilla-flavor history of your repository that you've become accustomed to:

```
commit 477e542bfa35942ddf069d85fbe3fb0923cfab47 (HEAD -> main)
Author: Chris Belanger <chris@razeware.com>
Date:   Wed Jan 23 16:49:56 2019 -0400

    Adding .gitignore files and HTML

commit ffcedc2397503831938894edffda5c5795c387ff
Author: Chris Belanger <chris@razeware.com>
Date:   Tue Jan 22 20:26:30 2019 -0400

    Adds all the good ideas about management
```

```
commit 84094274a447e76eb8f55def2c38b909ef94fa42
Author: Chris Belanger <chris@razeware.com>
Date:   Tue Jan 22 20:17:03 2019 -0400

    Removes terrible live streaming ideas

commit 67fd0aa99b5afc18b7c6cc9b4300a07e9fc88418
Author: Chris Belanger <chris@razeware.com>
Date:   Tue Jan 22 19:47:23 2019 -0400

    Moves platform ideas to website directory
```

This shows you a list of **ancestral commits** — that is, the set of commits that form the history of the current **head**. In this case, that's the most recent commit in the `main` branch of your repository. Press **Q** to exit this view.
The basic `git log` command shows you *all* the ancestral commits for this branch. What if you only wanted to see a few — say, three?

Limiting results

This is straightforward; simply execute the following command to show the number of commits you'd like to see, starting from the most recent:

```
git log -3
```

Git will then show you just the three most recent commits. You can replace the `3` in the above example to show any number of commits you'd prefer.
That's a little more manageable, but there's still a lot of detail in there. Wouldn't it be nice if there was a way to view *just* the commit messages and filter out all the other, extra information?
There is! Execute the following command to see a more compact view of the repository history:

```
git log --oneline
```

You'll see a quick, compact view of the commit history, which is arguably *far* more readable than the original output from `git log`:

```
~/GitApprentice/ideas $ git log --oneline
477e542 (HEAD -> main) Adding .gitignore files and HTML
ffcedc2 Adds all the good ideas about management
8409427 Removes terrible live streaming ideas
67fd0aa Moves platform ideas to website directory
0ddfac2 Updates book ideas for Symbian and MOS 6510
6c88142 Adding some tutorial ideas
.
.
.
```

This also shows you the **short hash** of a commit. Although you haven't looked at hashes in depth yet, there are long and short hashes for each commit that uniquely identify a commit within a repository.
For instance, if I take a look at the first line of the most recent commit on my repo with `git log -1` (that's the number "1", not the letter "l"), I see the following:

```
commit 477e542bfa35942ddf069d85fbe3fb0923cfab47 (HEAD -> main)
```

Now, to compare, I look at that same single commit with `git log -1 --oneline` (yes, you can stack multiple options with `git log`), I get the following:

```
477e542 (HEAD -> main) Adding .gitignore files and HTML
```

The short hash is simply the first seven characters of the long hash; in this case, `477e542`. For the average-sized development project, seven hexadecimal digits provides you with more than a quarter of a *billion* short hashes, so the possibility of hashes colliding between various commits is quite small.
When you ramp up to massively-sized Git repositories that live on for years, or even decades, the chance of two commits having the same hash becomes a reality.
Older versions of Git allowed you to configure the number of hash characters to use for your repository, but more recent versions of Git (from about 2017 onward) dynamically adapt this setting to suit the size of your project, so you don't usually have to worry about it.
Note: Are you wondering why some options to commands are preceded with a single dash, while others are preceded with double dashes?
This has its roots way back in the history of command-line-based operating systems. Generally, commands that have double dashes are the "long form" of a command, and are there for clarity.
For instance, the command `git log -p`, which you've used before, shows the diffs of your commits. But there's another command that only differs by the fact that the option is in uppercase: `git log -P`, which does something *entirely* different.
Since all these commands can get a bit confusing, especially where case matters, many modern command-line utilities provide long form alternatives to commands to be clearer about the the intent of a particular option.
In the above example, you can use `git log --patch` and `git log -p` interchangeably, because they mean *exactly* the same thing. The `--patch` option is more clear, but `-p` is more compact.

Graphical views of your repository

So what else can `git log` do? Well, Git has some simple methods to show you the branching history of your repository. Execute the following command to see a rather verbose view of the "tree" structure of your repository history:

```
git log --graph
```

Page through a few results by pressing the Spacebar (or scroll using the arrow keys), and you'll see where I merged a branch in an early version of the repository:

```
.
.
.
 commit fbc46d3d828fa57ef627742cf23e865689bf01a0
| Author: Chris Belanger <chris@razeware.com>
| Date:   Thu Jan 10 10:18:14 2019 -0400
|
|     Adding files for article ideas
|
*   commit 5fcdc0e77adc11e0b2beca341666e89611a48a4a
|\  Merge: 39c26dd cfbbca3
| | Author: Chris Belanger <chris@razeware.com>
| | Date:   Thu Jan 10 10:14:56 2019 -0400
| |
| |     Merge branch 'video_team'
| |
```

```
| * commit cfbbca371f4ecc80796a6c3fc0c084ebe181edf0
| | Author: Chris Belanger <chris@razeware.com>
| | Date:   Thu Jan 10 10:06:25 2019 -0400
| |
| |     Removing brain download as per ethics committee
.
.
.
```

And if you page down a little more, you'll see the point where I created the branch off of `master`:

```
* | commit 39c26dd9749eb627056b938313df250b669c1e4c
| | Author: Chris Belanger <chris@razeware.com>
| | Date:   Thu Jan 10 10:13:32 2019 -0400
| |
| |     I should write a book on git someday
| |
* | commit 43b4998d7bf0a6d7f779dd2c0fa4fe17aa3d2453
|/  Author: Chris Belanger <chris@razeware.com>
|   Date:   Thu Jan 10 10:12:36 2019 -0400
|
|       Adding book ideas file
|
* commit becd762cea13859ac32841b6024dd4178a706abe
| Author: Chris Belanger <chris@razeware.com>
| Date:   Thu Jan 10 09:49:23 2019 -0400
|
|     Creating the directory structure
|
* commit 73938223caa4ad5c3920a4db72920d5eda6ff6e1
  Author: crispy8888 <chris@razeware.com>
  Date:   Wed Jan 9 20:59:40 2019 -0400

      Initial commit
```

But that's still too much information. How could you collapse this tree-like view to only see the commit messages, but still see the branching history? That's right — by stacking the options to `git log`.
Exit by pressing the **Q** key and execute the following to see a more condensed view:

```
git log --oneline --graph
```

You'll see a nice, compact view of the history and branching structure:

```
~/GitApprentice/ideas $ git log --oneline --graph
* 477e542 (HEAD -> main) Adding .gitignore files and HTML
* ffcedc2 Adds all the good ideas about management
* 8409427 Removes terrible live streaming ideas
* 67fd0aa Moves platform ideas to website directory
* 0ddfac2 Updates book ideas for Symbian and MOS 6510
* 6c88142 Adding some tutorial ideas
* ce6971f Adding empty tutorials directory
* 57f31b3 Added new book entry and marked Git book complete
* f65a790 (origin/main, origin/HEAD) Updated README.md to reflect current working book title.
* c470849 (origin/master) Going to try this livestreaming thing
* 629cc4d Some scratch ideas for the iOS team
* fbc46d3 Adding files for article ideas
*   5fcdc0e Merge branch 'video_team'
|\
| * cfbbca3 Removing brain download as per ethics committee
| * c596774 Adding some video platform ideas
| * 06f468e Adding content ideas for videos
* | 39c26dd I should write a book on git someday
* | 43b4998 Adding book ideas file
|/
* becd762 Creating the directory structure
* 7393822 Initial commit
```

Viewing non-ancestral history

Git's not showing you the *complete* history, though. It's only showing you the history of things that have happened on the `main` branch. To tell Git to show you the complete history of everything it knows about, add the `--all` option to the previous command:

```
git log --oneline --graph --all
```

You'll see that there's an `origin/clickbait` branch off of `master` that Git wasn't telling you about earlier:

```
* 477e542 (HEAD -> main) Adding .gitignore files and HTML
* ffcedc2 Adds all the good ideas about management
* 8409427 Removes terrible live streaming ideas
* 67fd0aa Moves platform ideas to website directory
```

```
* 0ddfac2 Updates book ideas for Symbian and MOS 6510
* 6c88142 Adding some tutorial ideas
* ce6971f Adding empty tutorials directory
* 57f31b3 Added new book entry and marked Git book complete
* f65a790 (origin/main, origin/HEAD) Updated README.md to reflect current working book title.
* c470849 (origin/master) Going to try this livestreaming thing
* 629cc4d Some scratch ideas for the iOS team
| * e69a76a (origin/clickbait) Adding suggestions from Mic
| * 5096c54 Adding first batch of clickbait ideas
|/
* fbc46d3 Adding files for article ideas
*   5fcdc0e Merge branch 'video_team'
|\
| * cfbbca3 Removing brain download as per ethics committee
| * c596774 Adding some video platform ideas
| * 06f468e Adding content ideas for videos
* | 39c26dd I should write a book on git someday
* | 43b4998 Adding book ideas file
|/
* becd762 Creating the directory structure
* 7393822 Initial commit
```

Using Git shortlog

Git provides a very handy companion to `git log` in the form of `git shortlog`. This is a nice way to get a summary of the commits, perhaps for including in the release notes of your app. Sometimes “bug fixes and performance improvements” just isn’t quite enough detail, you know?
Execute the following command to see who’s made commits to this repository:

```
git shortlog
```

I see the following collection of commits for this repository:

```
Chris Belanger (18):
      Creating the directory structure
      Adding content ideas for videos
      Adding some video platform ideas
      Removing brain download as per ethics committee
      Adding book ideas file
      I should write a book on git someday
      Merge branch 'video_team'
      Adding files for article ideas
      Some scratch ideas for the iOS team
      Going to try this livestreaming thing
      Added new book entry and marked Git book complete
      Adding empty tutorials directory
      Adding some tutorial ideas
      Updates book ideas for Symbian and MOS 6510
      Moves platform ideas to website directory
      Removes terrible live streaming ideas
      Adds all the good ideas about management
      Adding .gitignore files and HTML

crispy8888 (1):
      Initial commit
.
.
.
```

I can see that I have 18 commits to this repository — and then there’s this `crispy8888` chap that created the initial repository. Well, that was nice of him. There are likely other changes from other users in there, including yourself.
You’ll notice that, in contrast to the standard `git log` command, `git shortlog` orders the commits in increasing time order. That makes more sense from a summary standpoint than showing everything in reverse-time order.
So far, you’ve seen how to use `git log` and `git shortlog` to give you a high-level view of the repository history with as much detail as you like. But sometimes you want to see a particular action in the repository. You know what you want to search for, but do you really have to scroll through all that output to retrieve what you’re looking for?
Git provides some excellent search functionality that you can use to find information about one particular file, or even particular changes across many files.

Searching Git history

Imagine that you wanted to see just the commits that this `crispy8888` fellow had made in the repository. Git gives you the ability to filter the output of `git log` to a particular author.
Execute the following command:

```
git log --author=crispy8888 --oneline
```

Git shows you the one change this fellow made:

```
7393822 Initial commit
```

If you want to search on a name that consists of two or more parts, simply enclose the name in quotation marks:

```
git log --author="Chris Belanger" --oneline
```

You can also search the commit messages of the repository, independent of who made the change.
Execute the following to find the commits, which have a commit message that contains the word “ideas”:

```
git log --grep=ideas --oneline
```

You should see something similar to the following:

```
ffcedc2 Adds all the good ideas about management
8409427 Removes terrible live streaming ideas
67fd0aa Moves platform ideas to website directory
0ddfac2 Updates book ideas for Symbian and MOS 6510
6c88142 Adding some tutorial ideas
629cc4d Some scratch ideas for the iOS team
fbc46d3 Adding files for article ideas
43b4998 Adding book ideas file
c596774 Adding some video platform ideas
06f468e Adding content ideas for videos
```

Note: Wondering what `grep` means? `grep` is a reference to a command line tool that stands for “global search regular expression and print”. `grep` is a wonderfully useful and powerful command line tool, and “grep” has come to be recognized in general usage as a verb that means “search,” especially in conjunction with regular expressions.

What if you’re interested in just a single file? That’s easy to do in Git.

Execute the following command to see all of the full commit messages for **books/book_ideas.md**:

```
git log --oneline books/book_ideas.md
```

You’ll see all the commits for just that file:

```
57f31b3 Added new book entry and marked Git book complete
39c26dd I should write a book on git someday
43b4998 Adding book ideas file
```

You can also see the commits that happened to the files in a particular directory:

```
git log --oneline books
```

This shows you all the changes that happened in that directory, but it’s not clear *which* files were changed.

To get a clearer picture of which files were changed in that directory, you can throw the `--stat` option on top of that command:

```
git log --oneline --stat books
```

This shows you the following details about the changes in this directory so that you can see what was changed, and even get a glimpse into how much was changed:

```
ffcedc2 Adds all the good ideas about management
 books/management_book_ideas.md | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
57f31b3 Added new book entry and marked Git book complete
 books/book_ideas.md | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
39c26dd I should write a book on git someday
 books/book_ideas.md | 1 +
 1 file changed, 1 insertion(+)
43b4998 Adding book ideas file
 books/book_ideas.md | 9 +++++++++
 1 file changed, 9 insertions(+)
becd762 Creating the directory structure
 books/.keep | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
```

You can also search the actual contents of the commit itself; that is, the changeset of the commit. This lets you look inside of your commits for particular words of interest or even whole snippets of code.

Find all of the commits in your code that deal with the term “Fortran” with the following command:

```
git log -S"Fortran"
```

You’ll see the following:

```
commit 43b4998d7bf0a6d7f779dd2c0fa4fe17aa3d2453
Author: Chris Belanger <chris@razeware.com>
Date:   Thu Jan 10 10:12:36 2019 -0400

    Adding book ideas file
```

There’s just the one commit, where the book ideas file was initially added. But, again, that’s not quite enough detail. Can you recall which option you can use to show the actual changes in the commit?

That’s right: It’s the `-p` option. Execute the command above, but this time, add the `-p` option to the end:

```
git log -S"Fortran" -p
```

You’ll see a bit more detail now:

```
commit 43b4998d7bf0a6d7f779dd2c0fa4fe17aa3d2453
Author: Chris Belanger <chris@razeware.com>
Date:   Thu Jan 10 10:12:36 2019 -0400

    Adding book ideas file

diff --git a/books/book_ideas.md b/books/book_ideas.md
new file mode 100644
index 0000000..f924368
```

```
--- /dev/null
+++ b/books/book_ideas.md
@@ -0,0 +1,9 @@
+# Ideas for new book projects
+
+- [ ] Hotubbing by tutorials
+- [x] Advanced debugging and reverse engineering
+- [ ] Animal husbandry by tutorials
+- [ ] Beginning tree surgery
+- [ ] CVS by tutorials
+- [ ] Fortran for fun and profit
+- [x] RxSwift by tutorials
```

That's better! You can now see the contents of that commit, where Git found the term "Fortran".
You've learned quite a lot about `git log` in this chapter, probably more than the average Git user knows. As you use Git more and more in your workflow, and as the history of your project grows from months to years, you'll find that `git log` will eventually be your best friend, and better at recalling things than your brain could ever be.

Challenges

Speaking of brains, why don't you exercise yours and reinforce the skills you learned in this chapter by taking on the four challenges of this chapter?

Challenge 1: Show all the details of commits that mark items as "done"

For this challenge, you need to find all of the commits where items have been ticked off as "done"; that is, ones that have an "x" inside the brackets, like so:
`[x]`
You'll need to search for the above string, and you'll need to use an option to not only show the basic commit details, but also show the contents of the changeset of the commit.

Challenge 2: Find all the commits with messages that mention "streaming"

You want to search through the commit **messages** to find where you or someone else has used the term "streaming" in the commit message itself, not necessarily in the content of the commit. Tip: What was that strangely named command you learned about earlier in this chapter?

Challenge 3: Get a detailed history of the videos directory

For this challenge, you need to show everything that's happened inside the **videos** directory, as far as Git's concerned. But, once again, the basic information about the commit is not enough. You also need to show the full details about that diff. So you'll tag a familiar option on to the end of the command… or can you?

Challenge 4: Find detailed information about all commits that contain "iOS 13"

In this final challenge, you need to find the commits whose diffs contain the term "iOS 13." This sounds similar to Challenge 1 above, but if you try to use the same command as you did in that challenge, you won't find any results. But trust me, there is at least one result in there. Tip: Did you remember to search "all" of the repository?

Key points

`git log` by itself shows a basic, vanilla view of the ancestral commits of the current `HEAD`.
`git log -p` shows the diff of a commit.
`git log -`*n* shows the last *n* commits.
`git log --oneline` shows a concise view of the short hash and the commit message.
You can stack options on `git log`, as in `git log -8 --oneline` to show the last 8 commits in a condensed form.
`git log --graph` shows a crude but workable graphical representation of your repository.
`git log --all` shows commits on other branches in the repository, not just the ancestors of the current `HEAD`.
`git shortlog` shows a summary of commits, grouped by their author them, in increasing time order.
`git log --author="<authorname>"` lets you search for commits by a particular author.
`git log --grep="<term>"` lets you search commit messages for a particular term.
`git log <path/to/filename>` will show you just the commits associated with that one file.
`git log <directory>` will show you the commits for files in a particular directory.
`git log --stat` shows a nice overview of the scope and scale of the change in each commit.
`git log -S"<term>"` lets you search the contents of a commit's changeset for a particular term.

Where to go from here?

You've learned a significant amount about how Git works under the hood, how commits work, how the staging area works, how to undo things you didn't mean to do, how to ignore files and how to leverage the power of `git log` to unravel the secrets of your repository.
But one thing you haven't yet really touched on is what makes Git so elegant and useful: its powerful branching model.
In fact, Git's branching mechanism is what sets it apart from most other version control systems, since it works extremely well with the way most developers go about their projects.
In the next chapter, you'll learn what `main` really means, how to create branches, how Git "thinks" about branches in your repository, the difference between local and remote repositories, how to switch branches, how to delete branches and more.

7. Branching 5. Ignoring Files in Git

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.