8
Merging Written by Bhagat Singh & Chris Belanger

Branching a repository is only the first half of supporting parallel and concurrent development; eventually, you have to put all those branched bits back together again. And, yes, that operation can be as complex as you think it might be!

Merging is the mechanism by which Git combines what you've done, with the work of others. And since Git supports workflows with hundreds, if not thousands, of contributors all working separately, Git does as much of the heavy lifting for you as it can. Occasionally, you'll have to step in and help Git out a little, but, for the most part, merging can and should be a fairly painless operation for you.

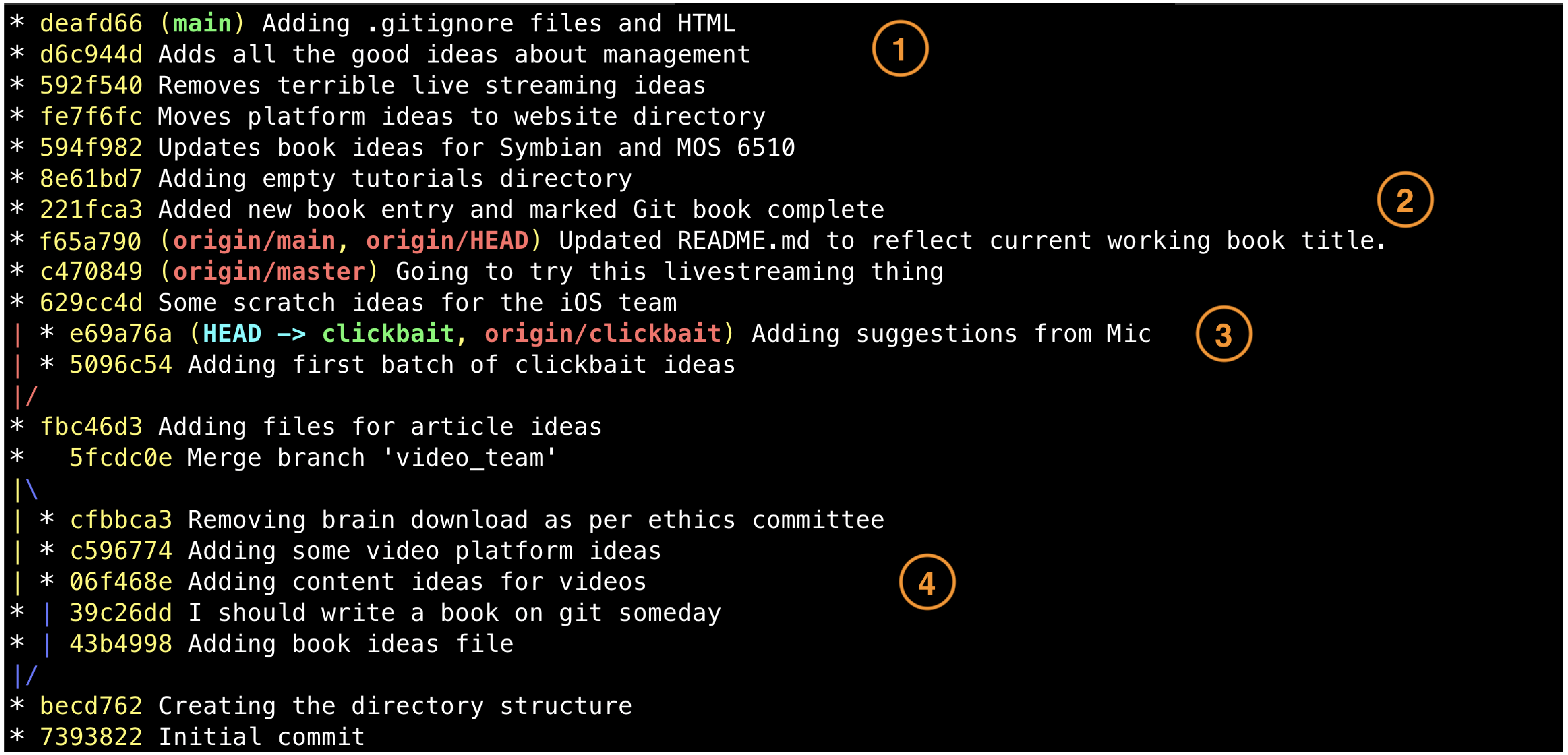
To begin this chapter, navigate to the **ideas** directory you've been working with through this book.

A look at your branches

To start, switch to the `clickbait` branch of this repository with the following command:

```
git checkout clickbait
```

If you were to visualize the branching history of your current **ideas** repository, with you sitting on the `clickbait` branch, it would look something like this :



```
* deafd66 (main) Adding .gitignore files and HTML
* d6c944d Adds all the good ideas about management
* 592f540 Removes terrible live streaming ideas
* fe7f6fc Moves platform ideas to website directory
* 594f982 Updates book ideas for Symbian and MOS 6510
* 8e61bd7 Adding empty tutorials directory
* 221fca3 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 (HEAD -> clickbait, 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
```

In the image above, you can see the following:

1. This is your local `main` branch. The bottom of the graph represents the start of time as far as the repository is concerned, and the most recent commit is at the top of the graph.
2. This is the `main` branch on `origin` — that is, the remote repository. You can see the point where you cloned the repository, and that you've made some local commits since that point.
3. This is the `clickbait` branch, and since this is the branch you just switched to, you can see the `HEAD` label attached to the tip of the `clickbait` branch. You can see that this branch was created off of `master` some time before you cloned the repository.
4. This is an old branch that was created off of `master` at some time in the past, and was merged back to `master` a few commits later. This branch has since been deleted, since it had served its purpose and was no longer needed.

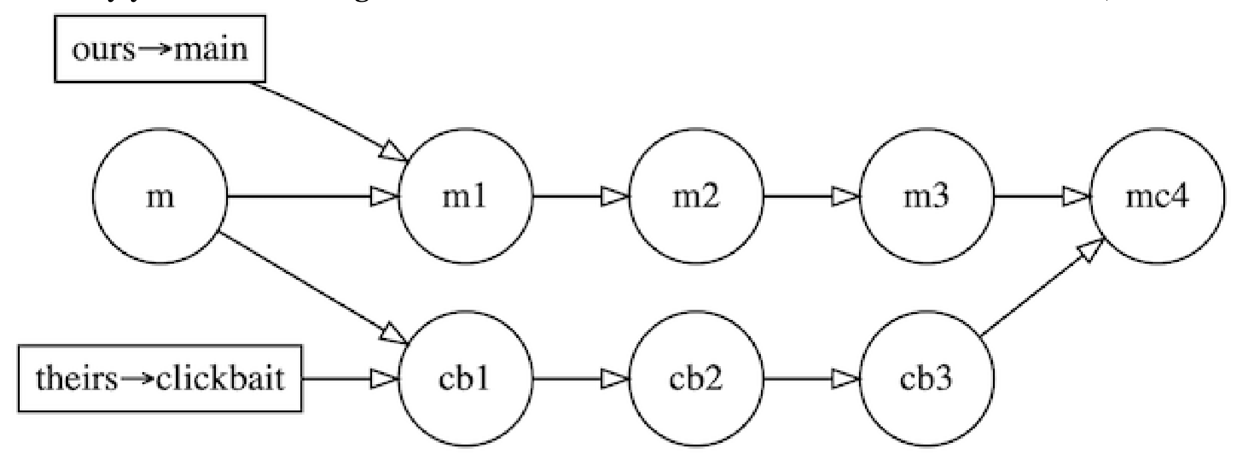
This is a fairly common development workflow; in a small team, `main` can effectively serve as the main development line, and developers make branches off of `main` to work on features or bug fixes, without messing with what's in the main development line. Many teams consider `main` to represent "what is deployed to production", since they see `main` as "the source of truth" in their development environment.

Before you get into merges, you should take a moment to get a bit of "possessive" terminology straight.

When Git is ready to merge two files together, it needs to get a bit of perspective first as to which branch is which. Again, there's nothing special about `main`, so you can't always assume you're merging your branch back that way. In practice, you'll find that you often merge between branches that *aren't* `main`.

So, therefore, Git thinks about branches in terms of **ours** and **theirs**. "Ours" refers to the branch to which you're merging back to, and "theirs" refers to the branch that you want to pull into "ours".

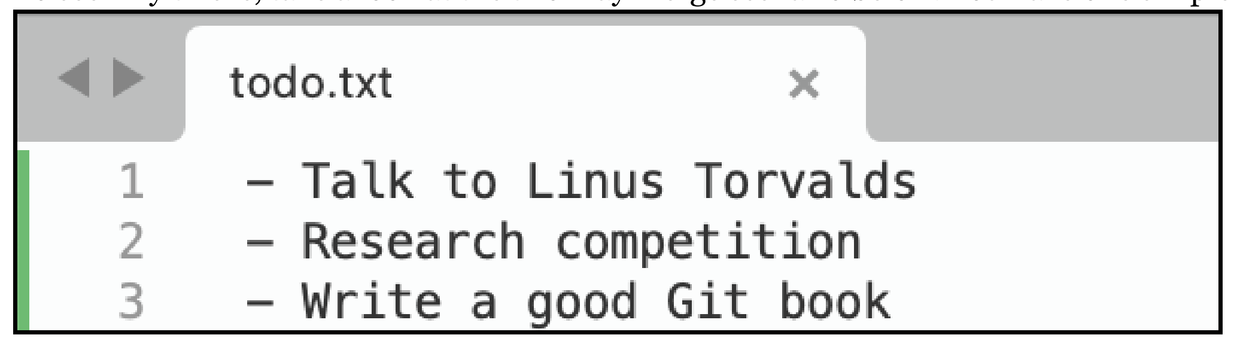
Let's say you want to merge the `clickbait` branch back into `main`. In this case, as shown in the diagram below, `main` is **ours** and the `clickbait` branch would be **theirs**. Keeping this distinction straight will help you immeasurably in your merging career.


$$\begin{tikzcd}
\boxed{\text{ours→main}} \arrow[dr] & & & & \\
m \arrow[r] \arrow[dr] & m1 \arrow[r] & m2 \arrow[r] & m3 \arrow[r] & mc4 \\
\boxed{\text{theirs→clickbait}} \arrow[r] & cb1 \arrow[r] & cb2 \arrow[r] & cb3 \arrow[ur] &
\end{tikzcd}$$

Three-way merges

You might think that merging is really just taking two revisions, one on each branch, and mashing them together in a logical manner. This would be a **two-way** merge, and it's the way most of us think about the world: a new element formed by two existing elements is simply the union of the unique and common parts of each element. However, a merge in Git actually uses *three* revisions to perform what is known as a **three-way merge**.

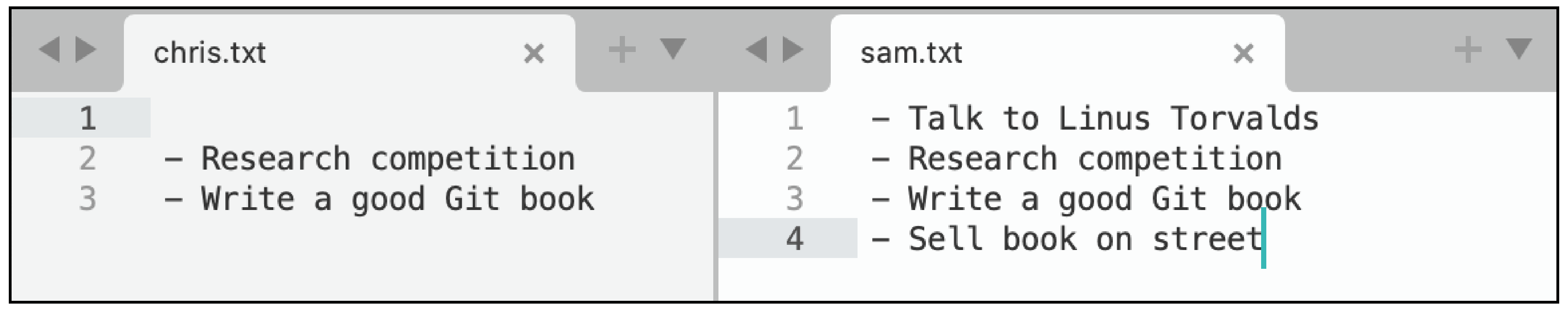
To see why this is, take a look at the two-way merge scenario below. You have one simple text file; you're working on one copy of the file while your friend is working on another, separate copy of that same file.



```
- Talk to Linus Torvalds
- Research competition
- Write a good Git book
```

The original file.

You delete a line from the top of the file, and your friend adds a line to the bottom of the file.



chris.txt

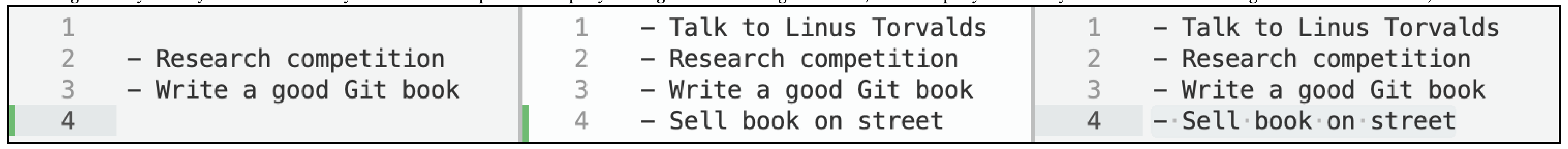
```
1
2  - Research competition
3  - Write a good Git book
```

sam.txt

```
1  - Talk to Linus Torvalds
2  - Research competition
3  - Write a good Git book
4  - Sell book on street
```

Chris' changes on the left; Sam's changes on the right.

Now imagine that you and your friend hand off your work to an impartial third party to merge this text file together. Now, this third party has literally no idea as to what the original state of this file was, so she has to make a guess as to what she should take from each file.



```
1
2  - Research competition
3  - Write a good Git book
4
```

```
1  - Talk to Linus Torvalds
2  - Research competition
3  - Write a good Git book
4  - Sell book on street
```

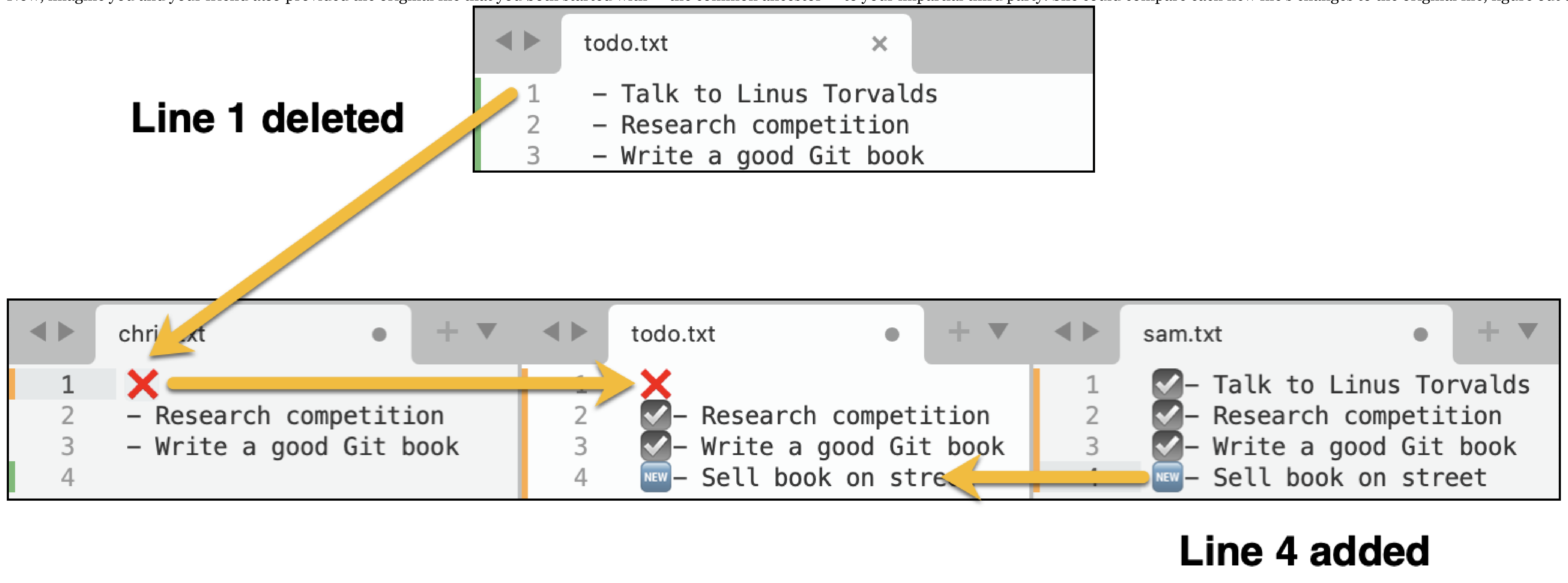
```
1  - Talk to Linus Torvalds
2  - Research competition
3  - Write a good Git book
4  - Sell book on street
```

With no background of what the starting point was, the person responsible to merge tries to preserve as many lines as possible in common to both files.

The end result is not quite what you intended, is it? You've ended up with all four lines; the impartial third party reviewer probably assumed Sam added a line to the top as well as a line to the bottom of Chris' work.

To perform an educated merge of these two files, your impartial third party has to know about the **common ancestor** of both of these files. This common ancestor is the third revision that comes in to play with a three-way merge.

Now, imagine you and your friend *also* provided the original file that you both started with — the common ancestor — to your impartial third party. She could compare each new file's changes to the original file, figure out the diff of your changes, figure out the diff of your friend's changes, and create the correct resulting merged document from the diffs of each.



todo.txt

```
1  - Talk to Linus Torvalds
2  - Research competition
3  - Write a good Git book
```

chris.txt

```
1  ❌
2  - Research competition
3  - Write a good Git book
4
```

todo.txt

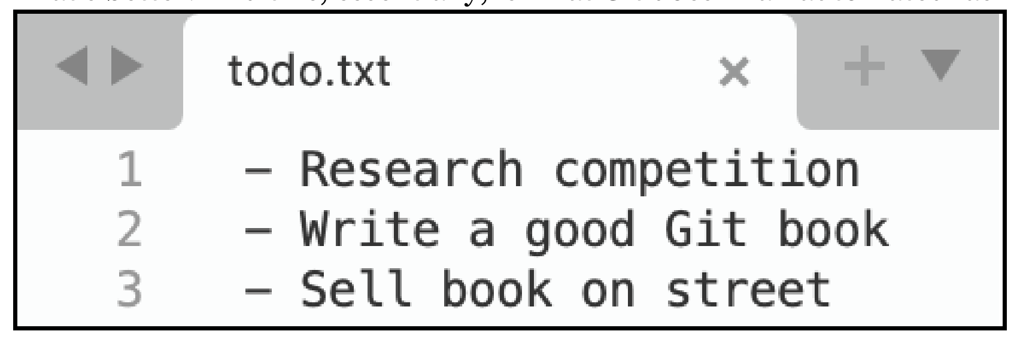
```
1  ❌
2  ✅- Research competition
3  ✅- Write a good Git book
4  🆕- Sell book on street
```

sam.txt

```
1  ✅- Talk to Linus Torvalds
2  ✅- Research competition
3  ✅- Write a good Git book
4  🆕- Sell book on street
```

Knowing the origin of each set of changes lets you detect that Line 1 was deleted by Chris, and Line 4 was added by Sam.

That's better. And this, essentially, is what Git does in an automated fashion. By performing three-way merges on your content, Git gets it right most of the time. Once in a while, Git won't be able to figure things out on its own, and you'll have to go in there and help it out a little bit. But you'll get into these scenarios a little later on in this book when you work on **merge conflicts**, which are a lot less scary than they sound.



todo.txt

```
1  - Research competition
2  - Write a good Git book
3  - Sell book on street
```

The result is what you both intended.

It's time for you to try out some merging yourself. Open up Terminal, navigate to the folder that houses your repository, and get ready to see how merging works in action.

Merging a branch

In this scenario, you're going to look at the work that someone else has made in the `clickbait` branch of the **ideas** repository, and merge those changes back into `main`.

Make sure you're on the `clickbait` branch by executing the following command (if you haven't already done this):

```
git checkout clickbait
```

Execute the following command to see what's been committed on this branch that you'll want to merge back to `main`:

```
git log clickbait --not main
```

This little gem is quite nice to keep on hand, as it tells you "what are the commits that are just in the `clickbait` branch, but not in `main`?" Just executing `git log` shows you *all* history of this branch, right back to the original creation of the `main` branch, which is too much information for your purposes.

You'll see the following output:

```
commit e69a76a6febf996a44a5de4dda6bde8569ef02bc (HEAD -> clickbait, origin/clickbait)
Author: Chris Belanger <chris@razeware.com>
Date:   Thu Jan 10 10:28:14 2019 -0400

    Adding suggestions from Mic

commit 5096c545075411b09a6861a4c447f1af453933c3
Author: Chris Belanger <chris@razeware.com>
Date:   Thu Jan 10 10:27:10 2019 -0400

    Adding first batch of clickbait ideas
```

Ok, there's two changes to merge back in; guess you'd better get cracking and merge these clickbait ideas before you lose any more traffic to your site.

To see the contents of the new file that's in this branch, execute the following command:

```
cat articles/clickbait_ideas.md
```

Some great ideas in there, for sure.

Recall that merging is the action of **pulling in changes** that have been done on another branch. In this case, you want to pull the changes from `clickbait` into the `main` branch. To do that, you'll have to be on the `main` branch first.

Execute the following to move to the `main` branch:

```
git checkout main
```

Now, what's in that **articles/clickbait_ideas.md** you looked at in the other branch? Execute that same command, again:

```
cat articles/clickbait_ideas.md
```

There's nothing in there. That's OK — you'll soon fill up that file with the ideas you're merging from the `clickbait` branch.

You're now back on the `main` branch, ready to pull in the changes from the `clickbait` branch. Execute the following command to merge the changes from `clickbait` to `main`:

```
git merge clickbait
```

Oh, heck, you're back in Vim. Well, at least Git has created a nice default message for you: `Merge branch 'clickbait`. That's enough detail for this merge, so simply accept this commit message and exit out:
Press **:** (colon) to enter Command mode.
Type **wq** and press Enter to write this file and quit the Vim editor.
As soon as you quit Vim, Git starts the merge operation for you and commits that merge, and it's likely done even before you know it.
Now, you can take a look at Git's graphical representation of the repository at this point with `git log --oneline --graph --all`:

```
*   55fb2dc (HEAD -> main) Merge branch 'clickbait'
|\
| * e69a76a (origin/clickbait, clickbait) Adding suggestions from Mic
| * 5096c54 Adding first batch of clickbait ideas
* | 477e542 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, origin/HEAD) 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
```

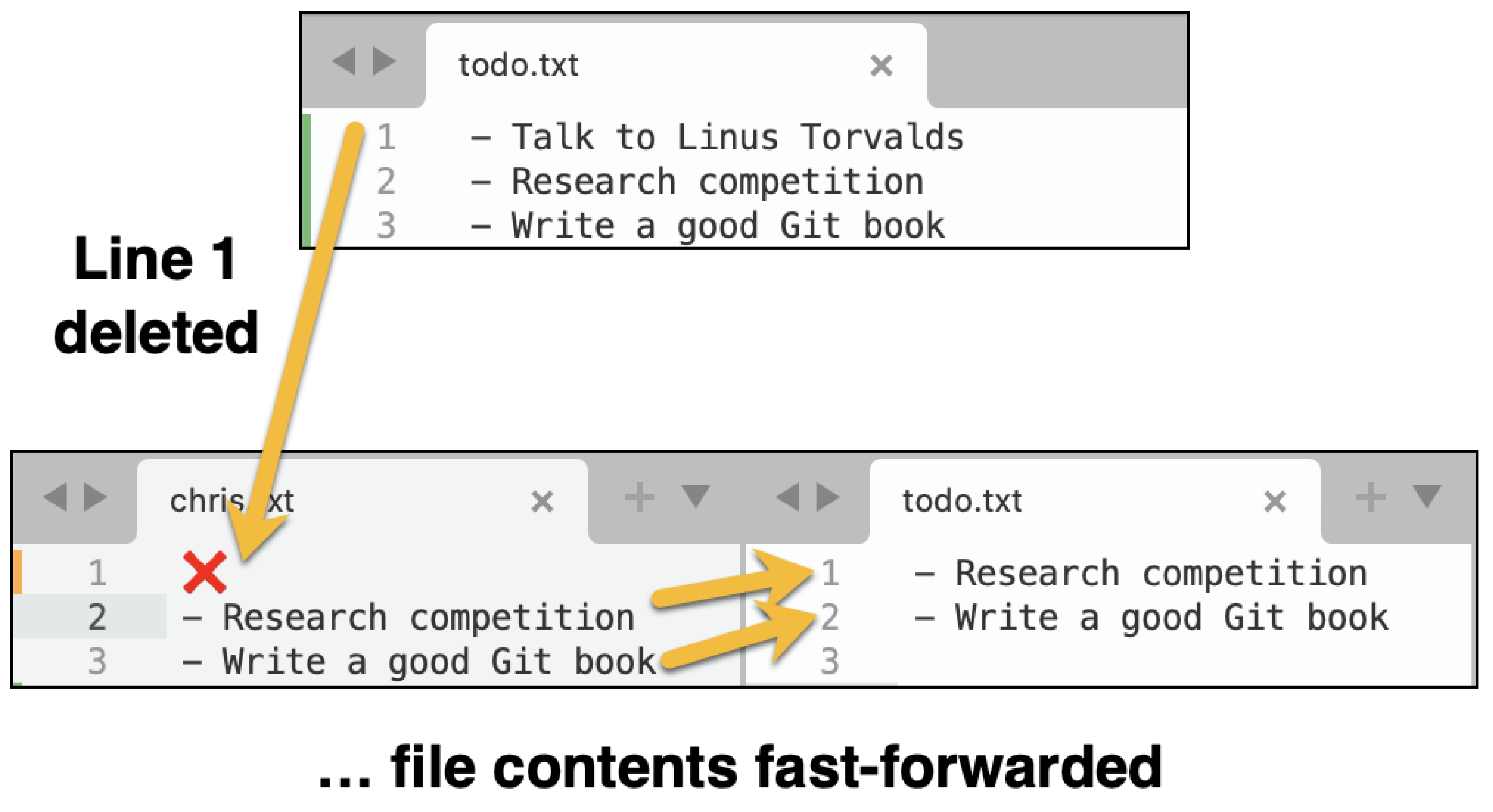
You can see at the top of the graph that Git has merged in your `clickbait` branch to `main` and that `HEAD` has now moved up to the latest revision, i.e., your merge commit.
If you want to prove that the file has now been brought into the `main` branch, execute the following command:

```
cat articles/clickbait_ideas.md
```

You'll see the contents of the file spat out to the console.

Fast-forward merge

There's another type of merge that happens in Git, known as the **fast-forward** merge. To illustrate this, think back to the example above, where you and your friend were working on a file. Your friend has gone away (probably hired away by Google or Apple, lucky sod), and you're now working on that file by yourself.
Once you've finished your revisions, you take your updated file, along with the original file (the common ancestor, again) to your impartial third party for merging. She's going to look at the common ancestor file, along with your new file, but she isn't going to see a third file to merge.
In this case, she's just going to commit your file on top of of the old file, because *there's nothing to merge*.



```
1  - Talk to Linus Torvalds
2  - Research competition
3  - Write a good Git book
```

```
1  ✕
2  - Research competition
3  - Write a good Git book
```

```
1  - Research competition
2  - Write a good Git book
3
```

If there are no other changes to the file to merge, Git simply commits your file over top of the original.
If no other person had touched the original file since you picked it up and started working on it, there's no real point in doing anything fancy, here. And while Git is far from lazy, it is terribly efficient and only does the work it absolutely needs to do to get the job done. This, in effect, is exactly what a fast-forward merge does.
To see this in action, you'll create a branch off of `main`, make a commit, and then merge the branch back to `main` to see how a fast-forward merge works.
First, execute the following to ensure you're on the `main` branch:

```
git checkout main
```

Now, create a branch named `readme-updates` to hold some changes to the README.md file:

```
git checkout -b readme-updates
```

Git creates that branch and automatically switches you to it. Now, open **README.md** in your favorite text editor, and add the following text to the end of the file:

```
This repository is a collection of ideas for articles, content and features at raywenderlich.com.

Feel free to add ideas and mark taken ideas as "done".
```

Save your changes, and return to Terminal. Stage your changes with the following command:

```
git add README.md
```

Now, commit that staged change with an appropriate message:

```
git commit -m "Adding more detail to the README file"
```

Now, to merge that change back to `main`. Remember — you need to be on the branch you want to pull the changes *into*, so you'll have to switch back to `main` first:

```
git checkout main
```

Now, before you merge that change in, take a look at Git's graph of the repository, using the `--all` flag to look on all branches, not just `main`:

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

Take a look at the top two lines of the result:

```
* 78eefc6 (readme-updates) Adding more detail to the README file
*   55fb2dc (HEAD -> main) Merge branch 'clickbait'
```

Git doesn't represent this as a fork in the branch — because it doesn't need to. Just as you saw in the example above with the single file, there's no need to merge anything, here. And that begs the question: If there's nothing to merge here, what will the resulting commit look like?
Time to find out! Execute the following command to merge `readme-updates` to `main`:

```
git merge readme-updates
```

Git tells you that it's done a fast-forward merge, right in the output:

```
~/GitApprentice/ideas $ git merge readme-updates
Updating 55fb2dc..78eefc6
Fast-forward
 README.md | 4 ++++
 1 file changed, 4 insertions(+)
```

You'll notice that Git didn't bring up the Vim editor, prompting you to add a commit message. You'll see why this is the case in just a moment. First, have a look at the resulting graph of the repository, using the command below:

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

Take a close look at the top two lines of the result. It looks like nothing much has changed, but take a look at where `HEAD` points now:

```
* 78eefc6 (HEAD -> main, readme-updates) Adding more detail to the README file
*   55fb2dc Merge branch 'clickbait' into main
```

Here, all Git has done is move the `HEAD` label to your latest commit. And this makes sense; Git isn't going to create a new commit if it doesn't have to. It's easier to just move the `HEAD` label along, since there's nothing to merge in this case. And *that's* why Git didn't prompt you to enter a commit message in Vim for this fast-forward merge.

Forcing merge commits

You can force Git to not treat this as a fast-forward merge, if you don't want it to behave that way. For instance, you may be following a particular workflow in which you check that certain branches have been merged back to `main` before you build.

But if those branches resulted in a fast-forward merge, for all intents and purposes, it will look like those changes were done directly on `main`, which isn't the case.

To force Git to create a merge commit when it doesn't really need to, all you need to do is add the `--no-ff` option to the end of your `merge` command. The challenge for this chapter will let you create a fast-forward situation, and see the difference between a merge commit and a fast-forward merge.

Note: Why wouldn't you always want a merge commit, especially if branching and merging are such cheap operations in Git? What's the point of moving `HEAD` along? Wouldn't it just be more clear to always have a merge commit?

This is a question that's just about as politically loaded as the age-old PC vs. Mac debate, the Android vs. iOS debate, or the cats vs. dogs debate (in which case, the answer is "dogs," if you were wondering).

This becomes particularly important on larger software projects with multiple contributors, where your commit history can have thousands upon thousands of commits over time. Merge commits can be seen as preserving the historical context of a feature or bugfix branch; it's clear that you branched, fixed, and then merged back in. Conversely, having lots of branches and merge commits — especially implicit merge commits, which you'll encounter later in this book — can make a repository's history harder to read and understand.

There's no real "right" answer, here; but don't believe people on the internet who claim that "merge commits are evil," because they're not. Git's job is to do its best to record what happened in your repository, and your workflow shouldn't necessarily have to change just to make sure that your commit history is linear and clean. However, you'll undoubtedly work with teams on both sides of the issue, so as long as you understand merge commits in Git, you'll do just fine, no matter which workflow your team champions.

Challenge

Challenge: Create a non-fast-forward merge

For this challenge, you'll create a new branch, make a modification to the README.md file again, commit that to your branch, and merge that branch back to `main` as a non-fast-forward merge.

This challenge will require the following steps:

. Ensure you're on the `main` branch.

. Create a branch named `contact-details`.

. Switch to that branch.

. Edit the README.md file and add the following text to the end of the file: "Contact: support@razeware.com".

. Save your edits to the file.

. Stage your changes.

. Commit your changes with an appropriate commit message, such as "Adding README contact information."

. Switch back to the `main` branch.

. Pull up the graph of the repository, and don't forget to use the `--all` option to see history of all branches. Make note of how `main` and `contact-details` look on this graph.

. Merge in the changes from `contact-details`, using the `--no-ff` option.

. Enter something appropriate in the merge message in Vim when prompted. Use the cheatsheet above to help you navigate through Vim if necessary.

. Pull up the graph of the repository again. How can you tell that this is a merge commit, and not a fast-forward commit?

If you get stuck, or want to check your solution, you can always find the answer to this challenge under the **challenge** folder for this chapter.

Key points

Merging combines work done on one branch with work done on another branch.
Git performs **three-way merges** to combine content.
Ours refers to the branch to which you want to pull changes into; **theirs** refers to the branch that has the changes you want to pull into **ours**.
`git log <theirs> --not <ours>` shows you what commits are on the branch you want to merge, that aren't in your branch already.
`git merge <theirs>` merges the commits on the "theirs" branch into "our" branch.
Git automatically creates a merge commit message for you, and lets you edit it before continuing with the merge.
A **fast-forward** merge happens when there have been no changes to "ours" since you branched off "theirs", and results in no merge commit being made.
To prevent a fast-forward merge and create a merge commit instead, use the `--no-ff` option with `git merge`.

Where to go from here?

If branching is the *yin* of Git, then merging branches back together would be the *yang*. Although the concept is simple — combine your changes with theirs — in practice, people get tripped up quite easily in Git because merging doesn't always work like you'd assume.

The next chapter, Syncing with a Remote, takes you beyond your local environment, and shows you how to synchronize your local changes with what's up on the server.

9. Syncing With a Remote 7. Branching

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.