git --fast-version-control

Search entire site...

Download this book in PDF, mobi, or ePub form for free.

This book is translated into Deutsch, 简体中文, 正體中文, Français, 日本語, Nederlands, Русский, 한국어, Português (Brasil) and Čeština.

Partial translations available in Arabic, Español, Indonesian, Italiano, Suomi, Македонски, Polski and Türkçe.

Translations started for Azərbaycan dili, Беларуская, Català, Esperanto, Español (Nicaragua), فارسی, हिन्दी, Magyar, Norwegian Bokmål, Română, Српски, ภาษาไทย, Tiếng Việt and Українська.

The source of this book and the translations are hosted on GitHub.
Patches, suggestions, and comments are welcome.

Related StackOverflow Questions

- About Git's merge and rebase **5** votes / **3** Answers
- git rebase vs git merge **81** votes / **5** Answers
- How to do a rebase with git gui? **5** votes / **3** Answers
- recovering from git rebase **27** votes / **15** Answers
- git pull VS git fetch git rebase **15** votes / **3** Answers
- Undoing a git rebase **160** votes / **7** Answers
- How to know if there is a git rebase in progress? **16** votes / **7** Answers

Chapters ▾

3.6 Git Branching - Rebasing

Rebasing

In Git, there are two main ways to integrate changes from one branch into another: the `merge` and the `rebase`. In this section you'll learn what rebasing is, how to do it, why it's a pretty amazing tool, and in what cases you won't want to use it.

The Basic Rebase

If you go back to an earlier example from the Merge section (see Figure 3-27), you can see that you diverged your work and made commits on two different branches.

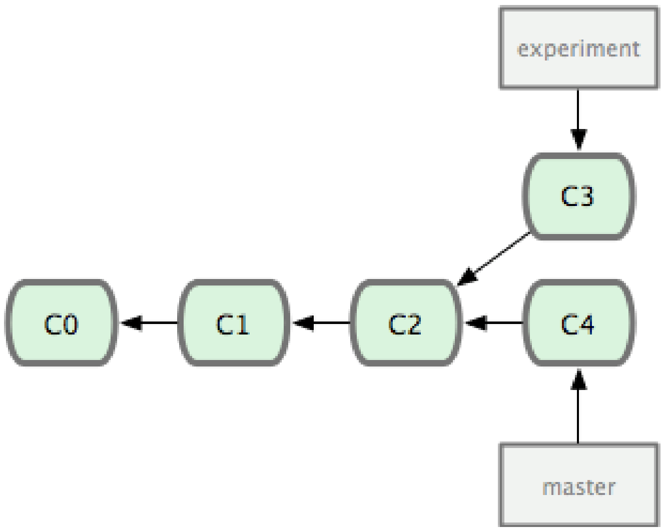

Figure 3-27. Your initial diverged commit history.

The easiest way to integrate the branches, as we’ve already covered, is the `merge` command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit), as shown in Figure 3-28.

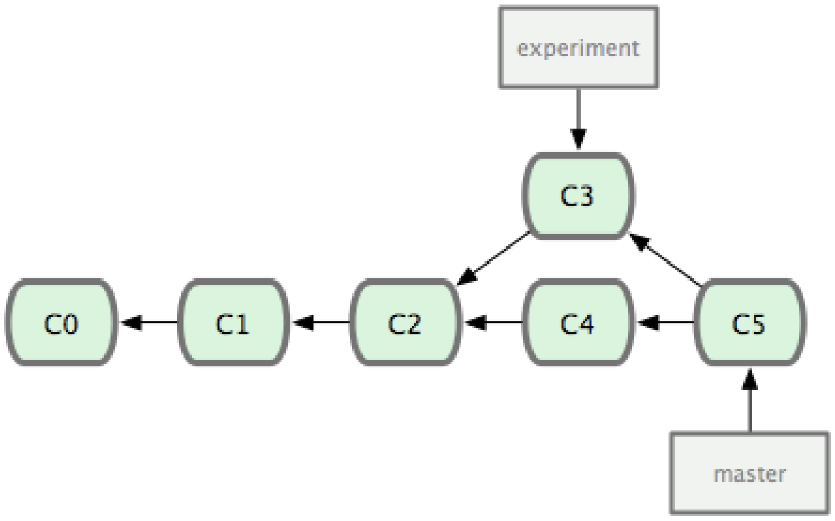

Figure 3-28. Merging a branch to integrate the diverged work history.

However, there is another way: you can take the patch of the change that was introduced in C3 and reapply it on top of C4. In Git, this is called *rebasing*. With the `rebase` command, you can take all the changes that were committed on one branch and replay them on another one.

In this example, you’d run the following:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
```

```
Applying: added staged command
```

It works by going to the common ancestor of the two branches (the one you’re on and the one you’re rebasing onto), getting the diff introduced by each commit of the branch you’re on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn. Figure 3-29 illustrates this process.

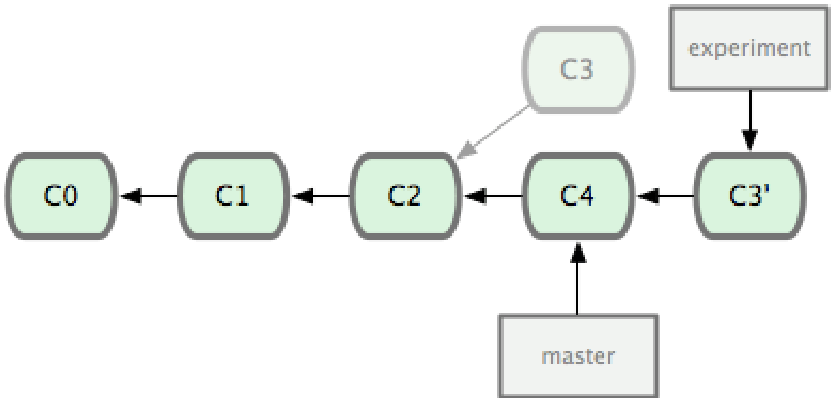

Figure 3-29. Rebasing the change introduced in C3 onto C4.

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

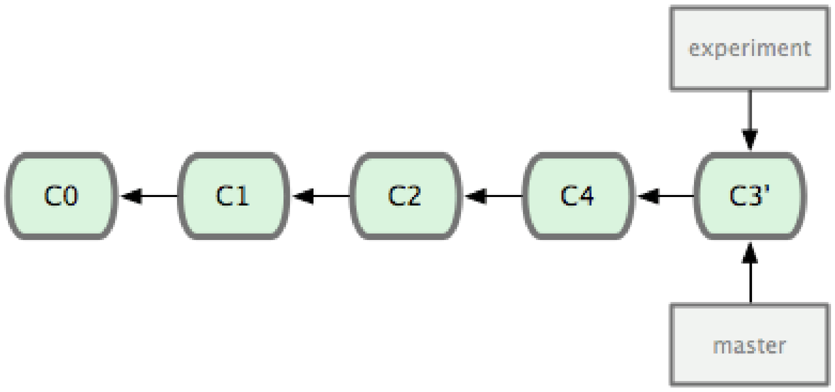

Figure 3-30. Fast-forwarding the master branch.

Now, the snapshot pointed to by C3' is exactly the same as the one that was pointed to by C5 in the merge example. There is no difference in the end product of the integration, but rebasing makes for a cleaner history. If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel.

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

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

More Interesting Rebases

You can also have your rebase replay on something other than the rebase branch. Take a history like Figure 3-31, for example. You branched a topic branch (`server`) to add some server-side functionality to your project, and made a commit. Then, you branched off that to make the client-side changes (`client`) and committed a few times. Finally, you went back to your server branch and did a few more commits.

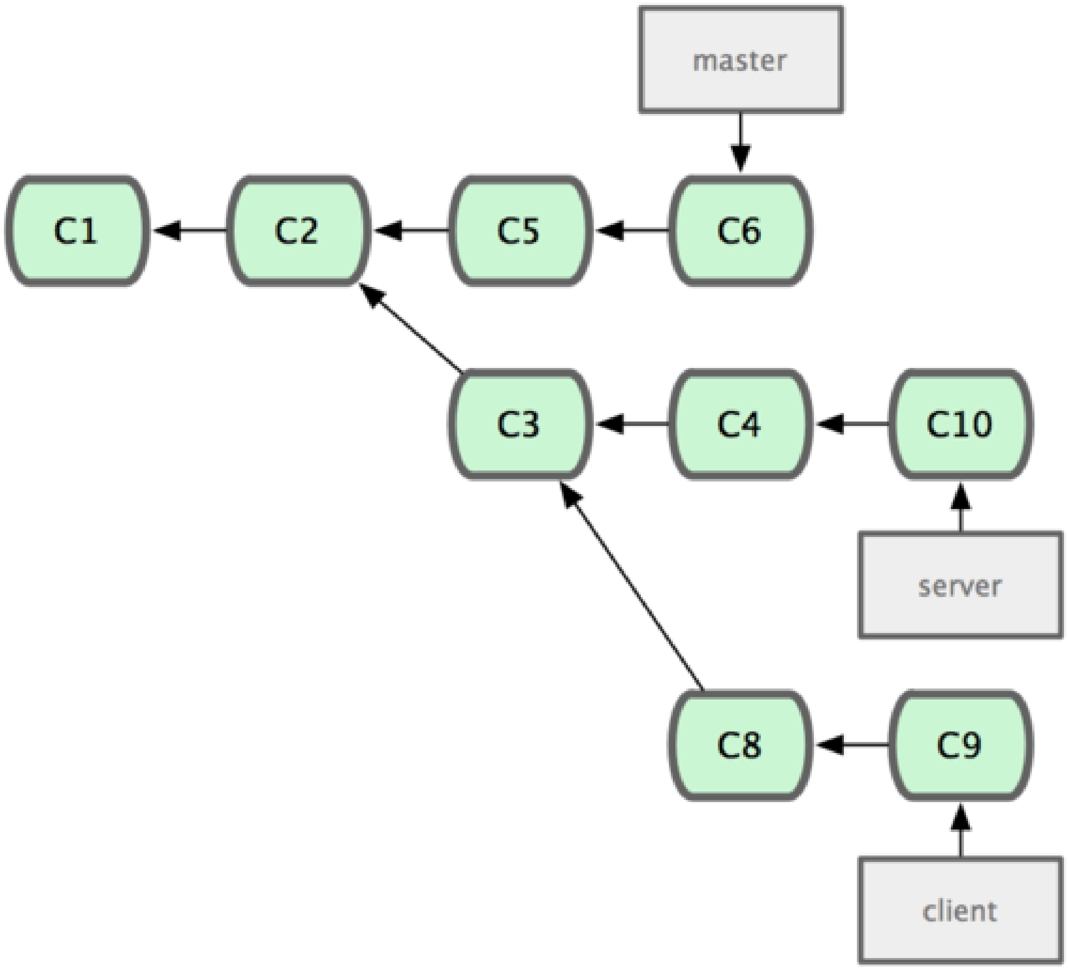

Figure 3-31. A history with a topic branch off another topic branch.

Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes until it's tested further. You can take the changes on client that aren't on server (C8 and C9) and replay them on your master branch by using the `--onto` option of `git rebase`:

```
$ git rebase --onto master server client
```

This basically says, "Check out the client branch, figure out the patches from the common ancestor of the `client` and `server` branches, and then replay them onto `master`." It's a bit complex; but the result, shown in Figure 3-32, is pretty cool.

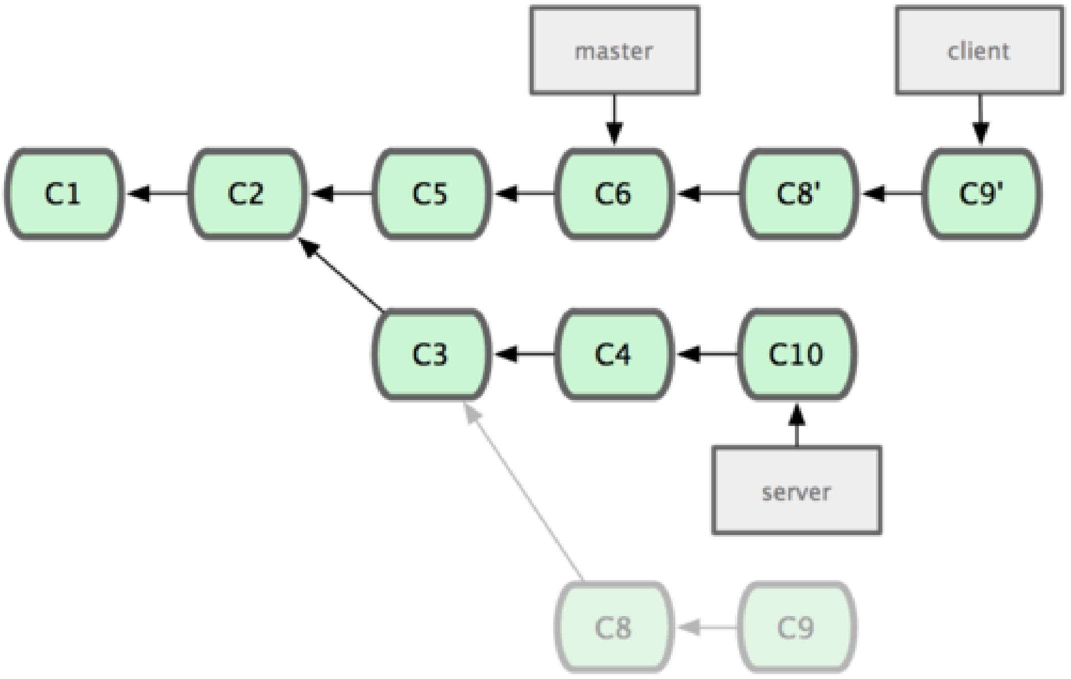

Figure 3-32. Rebasing a topic branch off another topic branch.

Now you can fast-forward your master branch (see Figure 3-33):

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

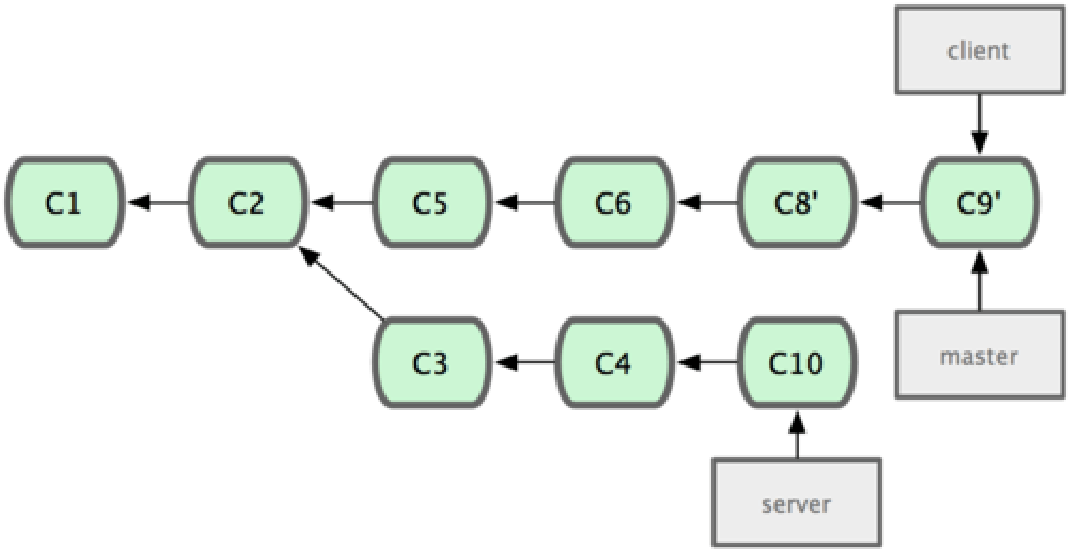

Figure 3-33. Fast-forwarding your master branch to include the client branch changes.

Let’s say you decide to pull in your server branch as well. You can rebase the server branch onto the master branch without having to check it out first by running `git rebase [basebranch] [topicbranch]` — which checks out the topic branch (in this case, `server`) for you and replays it onto the base branch (`master`):

```
$ git rebase master server
```

This replays your `server` work on top of your `master` work, as shown in Figure 3-34.

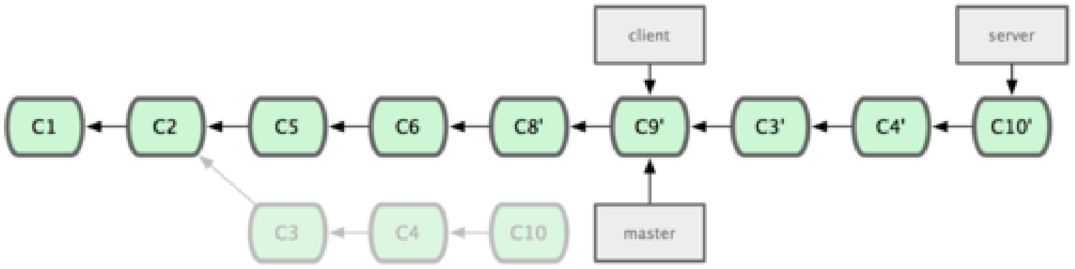

Figure 3-34. Rebasing your server branch on top of your master branch.

Then, you can fast-forward the base branch (`master`):

```
$ git checkout master
$ git merge server
```

You can remove the `client` and `server` branches because all the work is integrated and you don't need them anymore, leaving your history for this entire process looking like Figure 3-35:

```
$ git branch -d client
$ git branch -d server
```

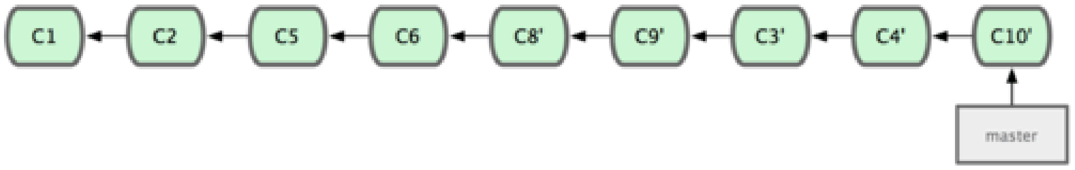

Figure 3-35. Final commit history.

The Perils of Rebasing

Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

Do not rebase commits that you have pushed to a public repository.

If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family.

When you rebase stuff, you're abandoning existing commits and creating new ones that are similar but different. If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with `git rebase` and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

Let's look at an example of how rebasing work that you've made public can cause problems. Suppose you clone from a central server and then do some work off that. Your commit history looks like Figure 3-36.

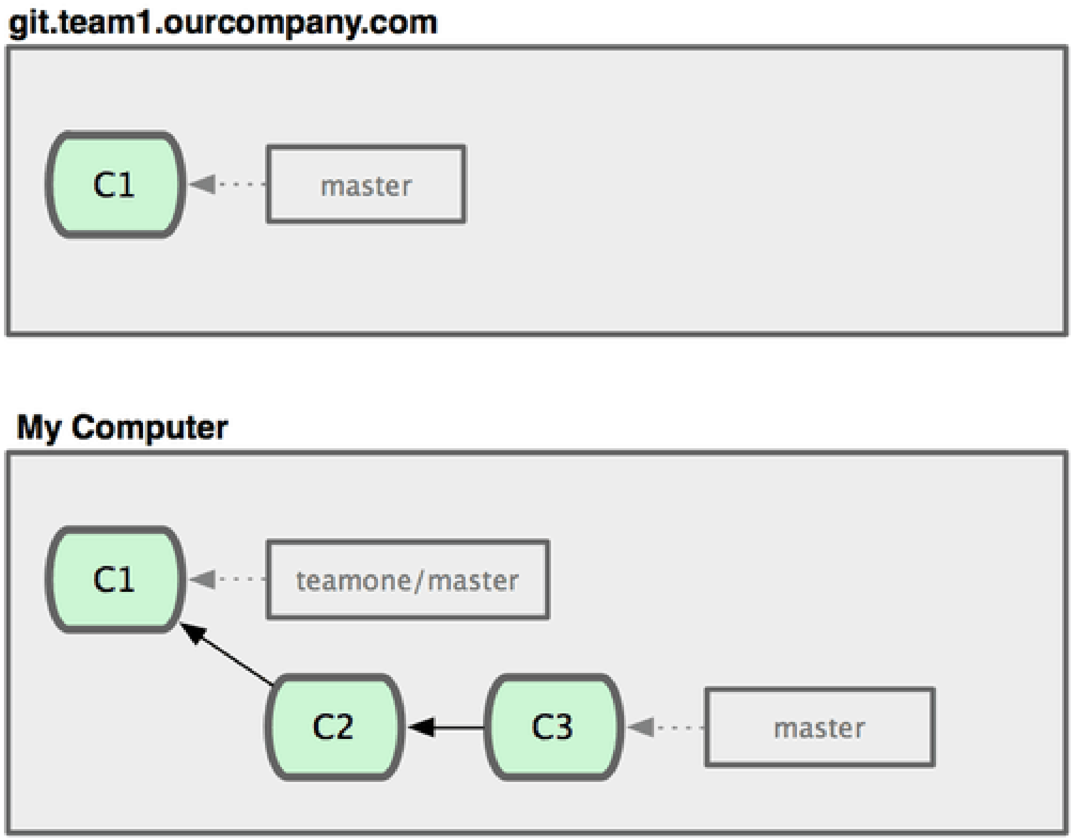

Figure 3-36. Clone a repository, and base some work on it.

Now, someone else does more work that includes a merge, and pushes that work to the central server. You fetch them and merge the new remote branch into your work, making your history look something like Figure 3-37.

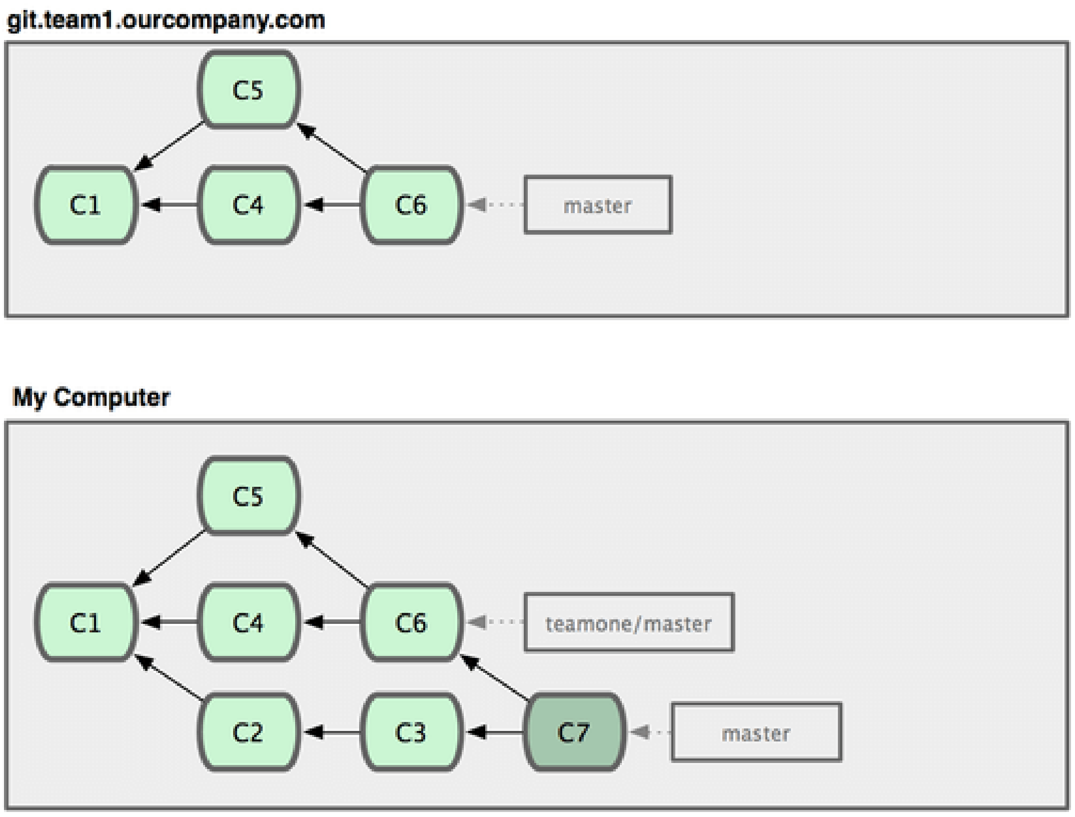

Figure 3-37. Fetch more commits, and merge them into your work.

Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a `git push --force` to overwrite the history on the server. You then fetch from that server, bringing down the new commits.

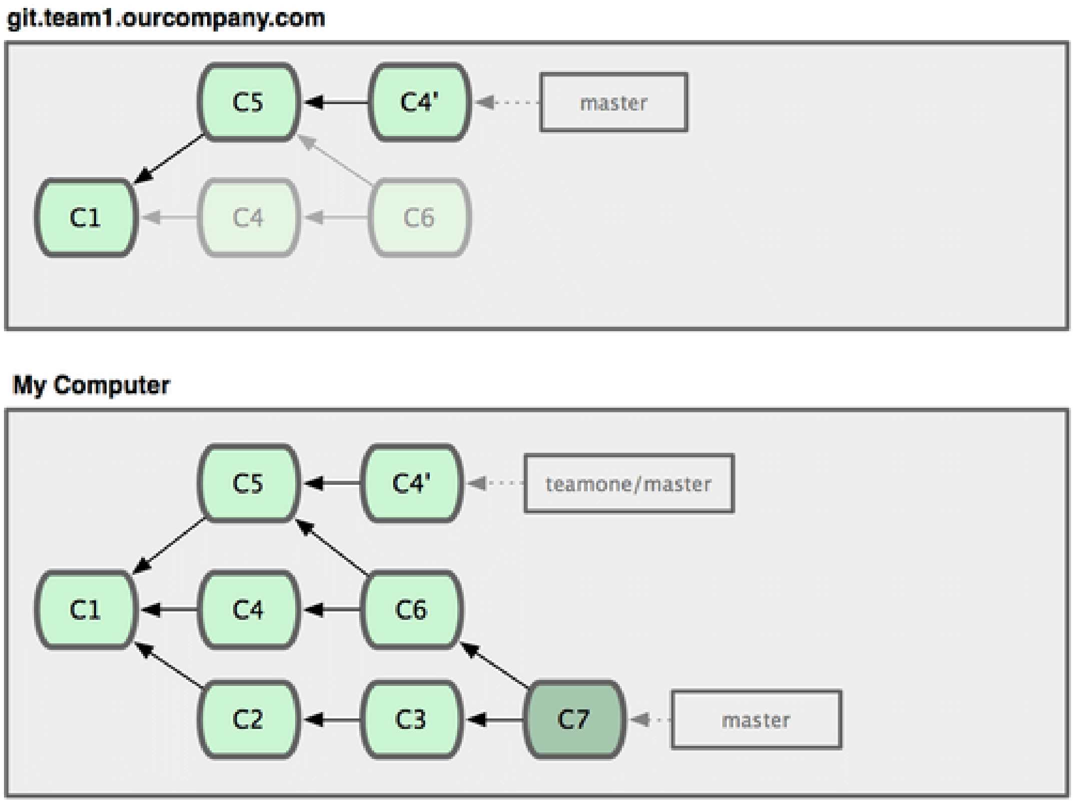

Figure 3-38. Someone pushes rebased commits, abandoning commits you've based your work on.

At this point, you have to merge this work in again, even though you've already done so. Rebasing changes the SHA-1 hashes of these commits so to Git they look like new commits, when in fact you already have the C4 work in your history (see Figure 3-39).

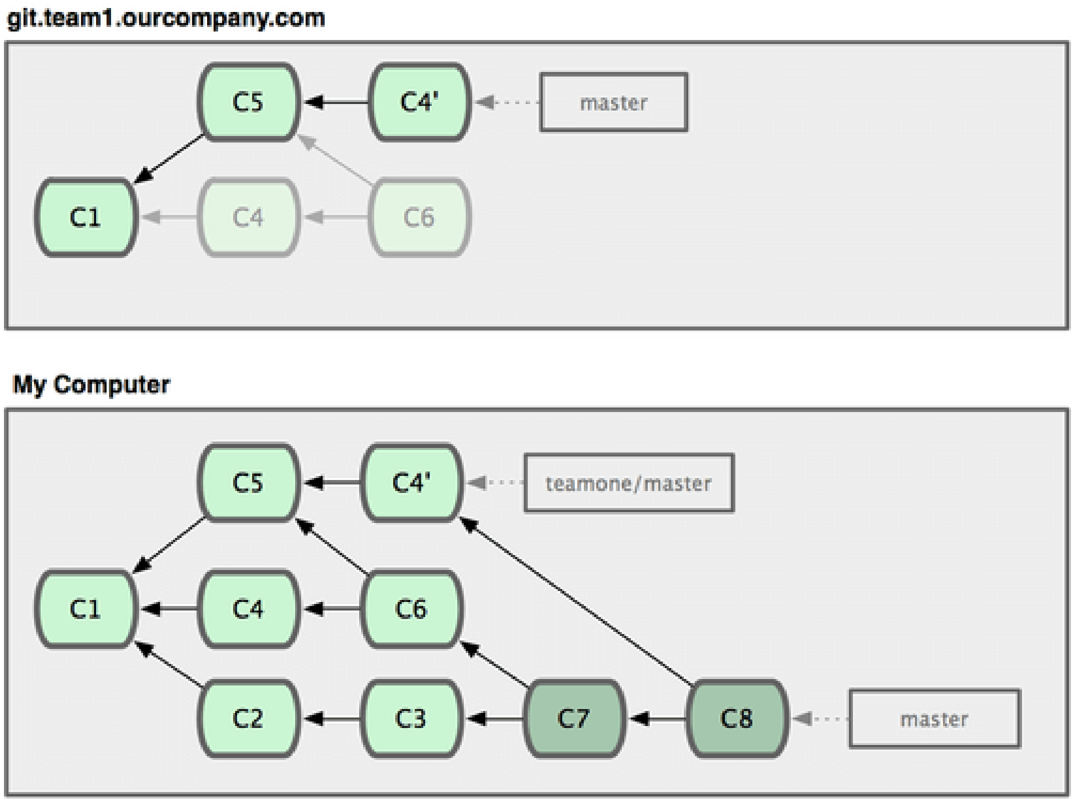

Figure 3-39. You merge in the same work again into a new merge commit.

You have to merge that work in at some point so you can keep up with the other developer in the future. After you do that, your commit history will contain both the C4 and C4' commits, which have different SHA-1 hashes but introduce the same work and have the same commit message. If you run a `git log` when your history looks like this, you'll see two commits that have the same author date and message, which will be confusing. Furthermore, if you push this history back up to the server, you'll reintroduce all those rebased commits to the central server, which can further confuse people.

If you treat rebasing as a way to clean up and work with commits before you push them, and if you only rebase commits that have never been available publicly, then you'll be fine. If you rebase commits that have already been pushed publicly, and people may have based work on those commits, then you may be in for some frustrating trouble.

prev | next
This open sourced site is hosted on GitHub.
Patches, suggestions, and comments are welcome.
Git is a member of Software Freedom Conservancy