Chapter 3. Git Branching

- Branching means you diverge from the main line of development and continue to do work without messing with that main line.
- The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast.

Branches in a Nutshell

- Git doesnt store data as a series of changesets or differences, but instead as a series of snapshots.
- When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged.
- This object also contains the authors name and email, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents).

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

- Git checksums each subdirectory (in this case, just the root project directory) and stores those tree objects in the Git repository.

- Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.
- Your Git repository now contains five objects: one blob for the contents of each of your three files, one tree that lists the contents of the directory and specifies which file names are stored as which blobs, and one commit with the pointer to that root tree and all the commit metadata.

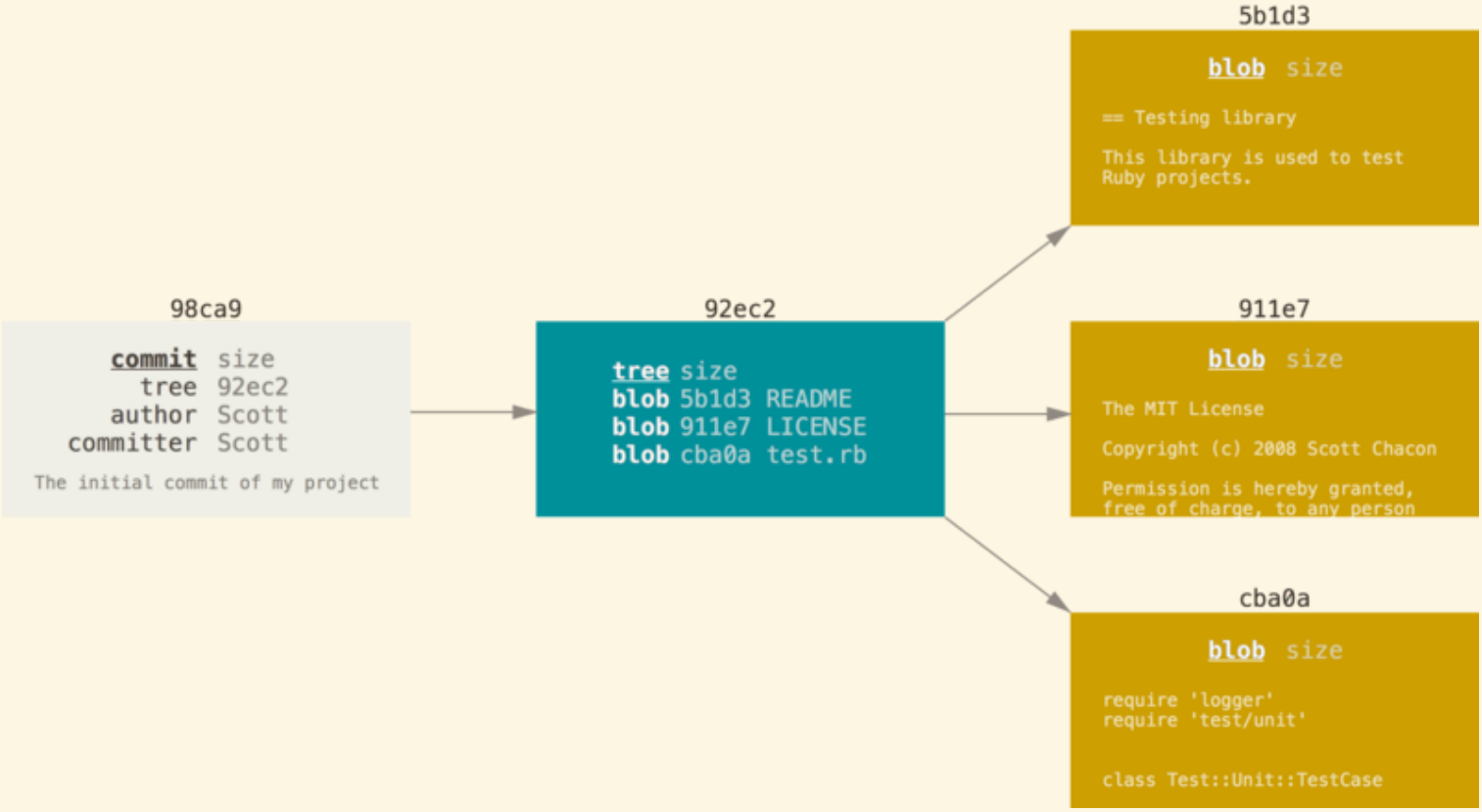


Figure 3-1. A commit and its tre.

- If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.

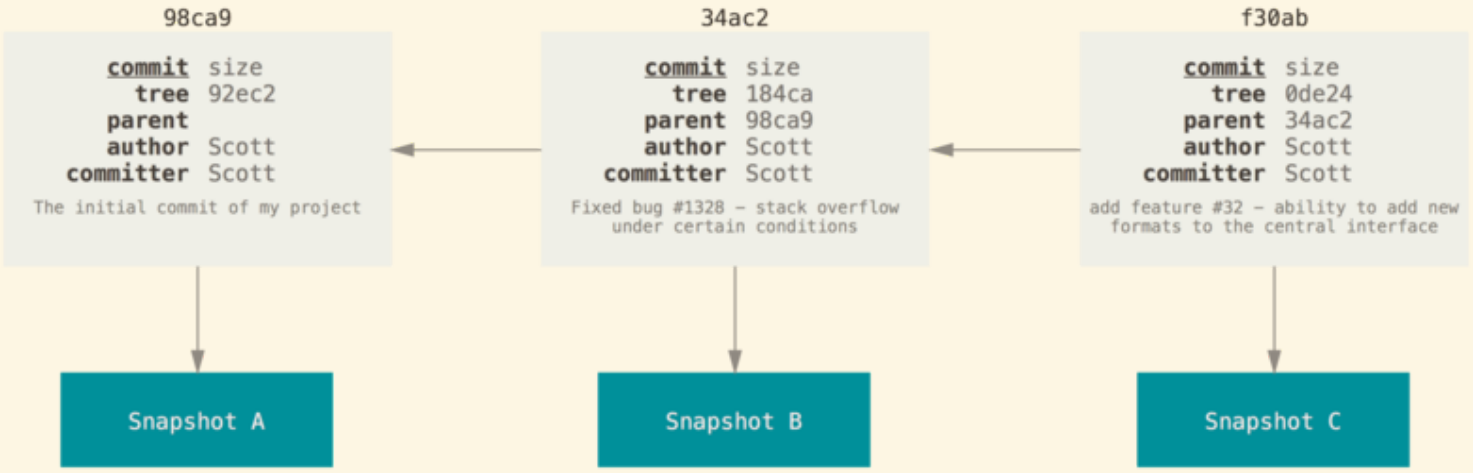


Figure 3-2. Commits and their parent.

- A branch in Git is simply a lightweight movable pointer to one of these commits.
- The default branch name in Git is `master`.

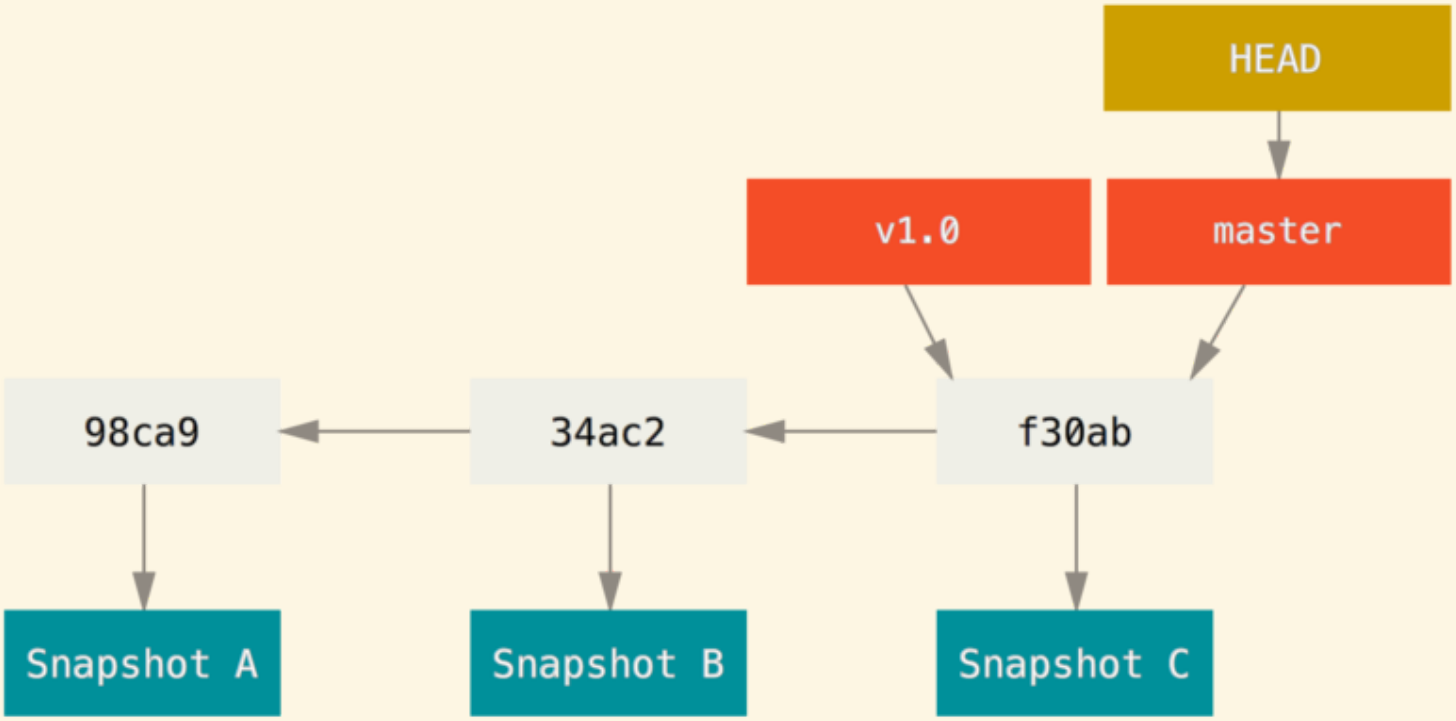


Figure 3-3. A branch and its commit histor.

Creating a New Branch

```
$ git branch testing
```

- This creates a new pointer at the same commit youre currently on.

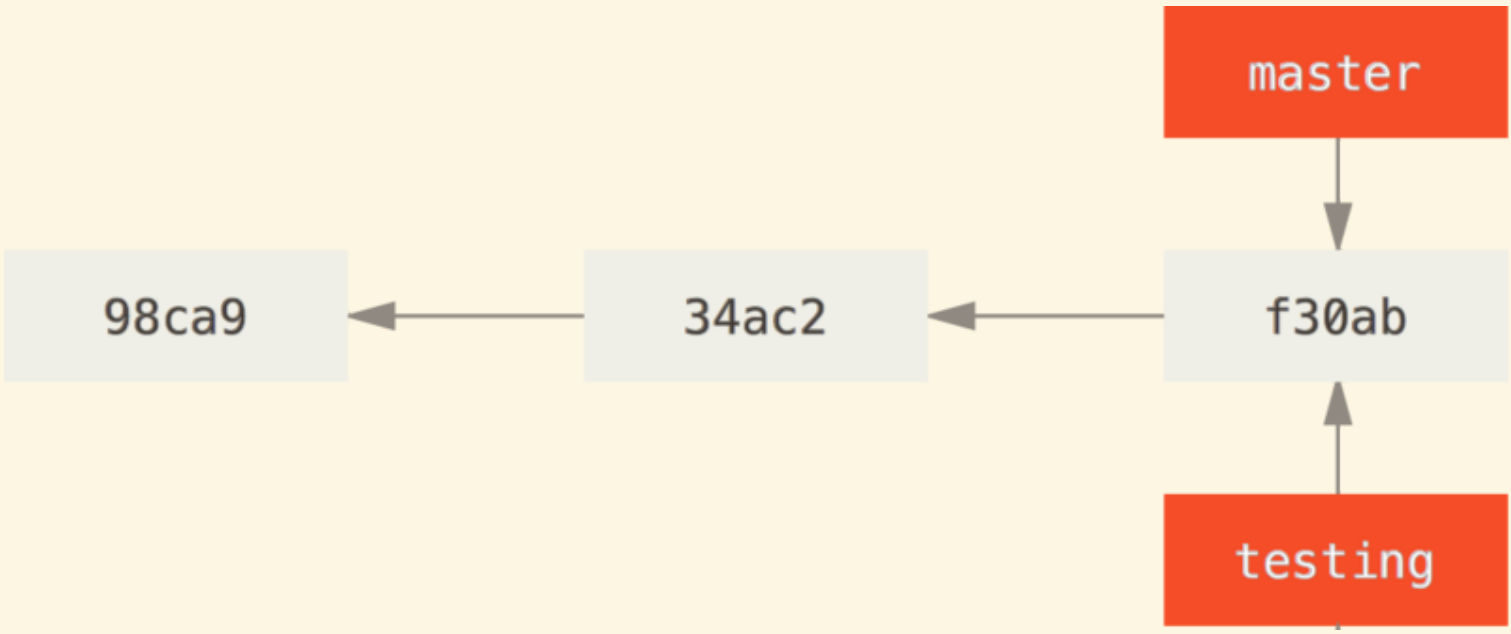


Figure 3-4. Two branches pointing into the same series of commit.

- branch currently on? It keeps a special pointer called `HEAD`.
- This is a pointer to the local branch youre currently on.
- The `git branch` command only *created* a new branch - it didnt switch to that branch.

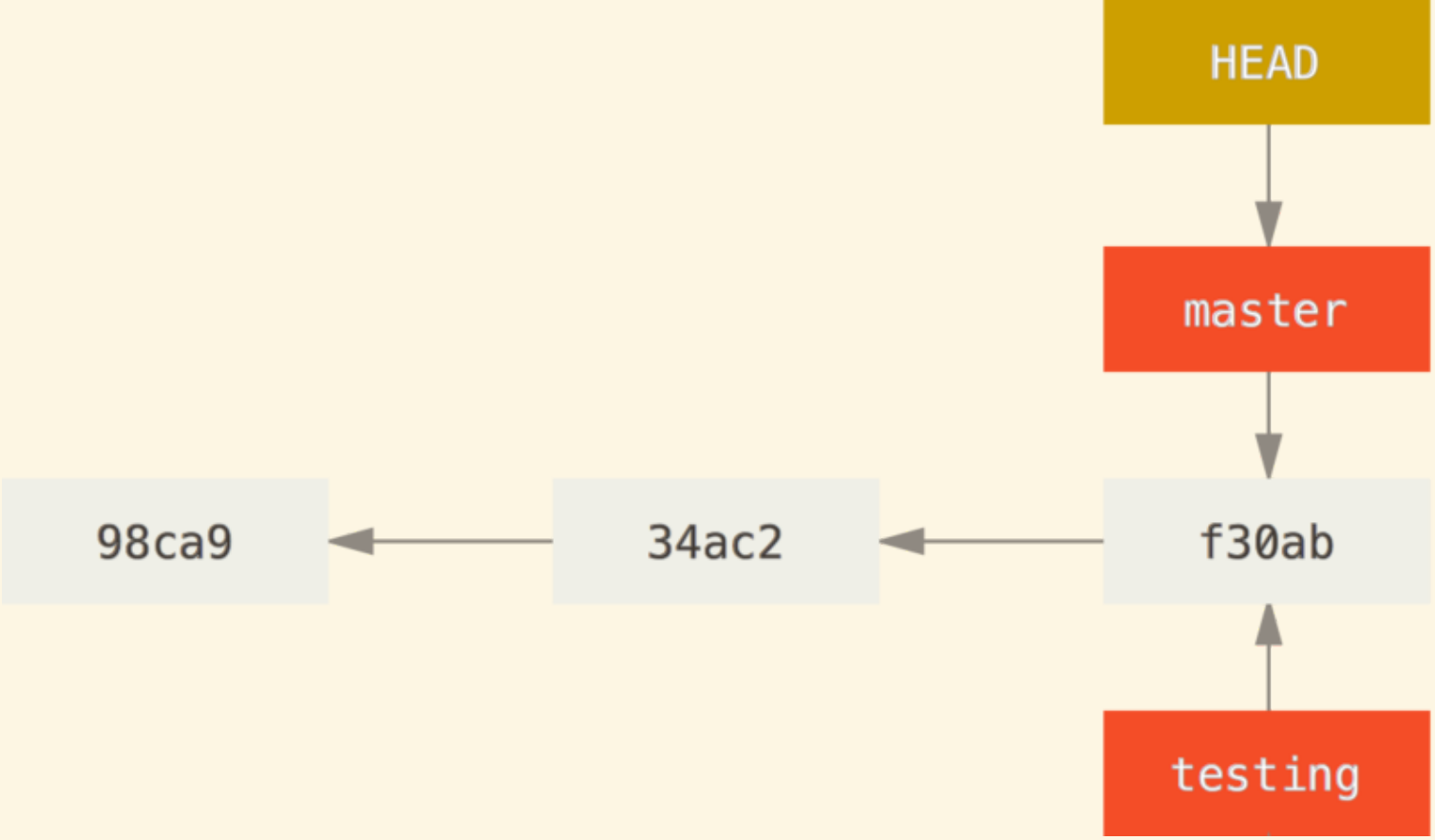


Figure 3-5. HEAD pointing to a branc.

```
$ git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

- You can see the master and testing branches that are right there next to the `f30ab` commit.

Switching Branches

- To switch to an existing branch, you run the `git checkout` command. Lets switch to the new testing branch:

```
$ git checkout testing
```

- This moves `HEAD` to point to the `testing` branch.

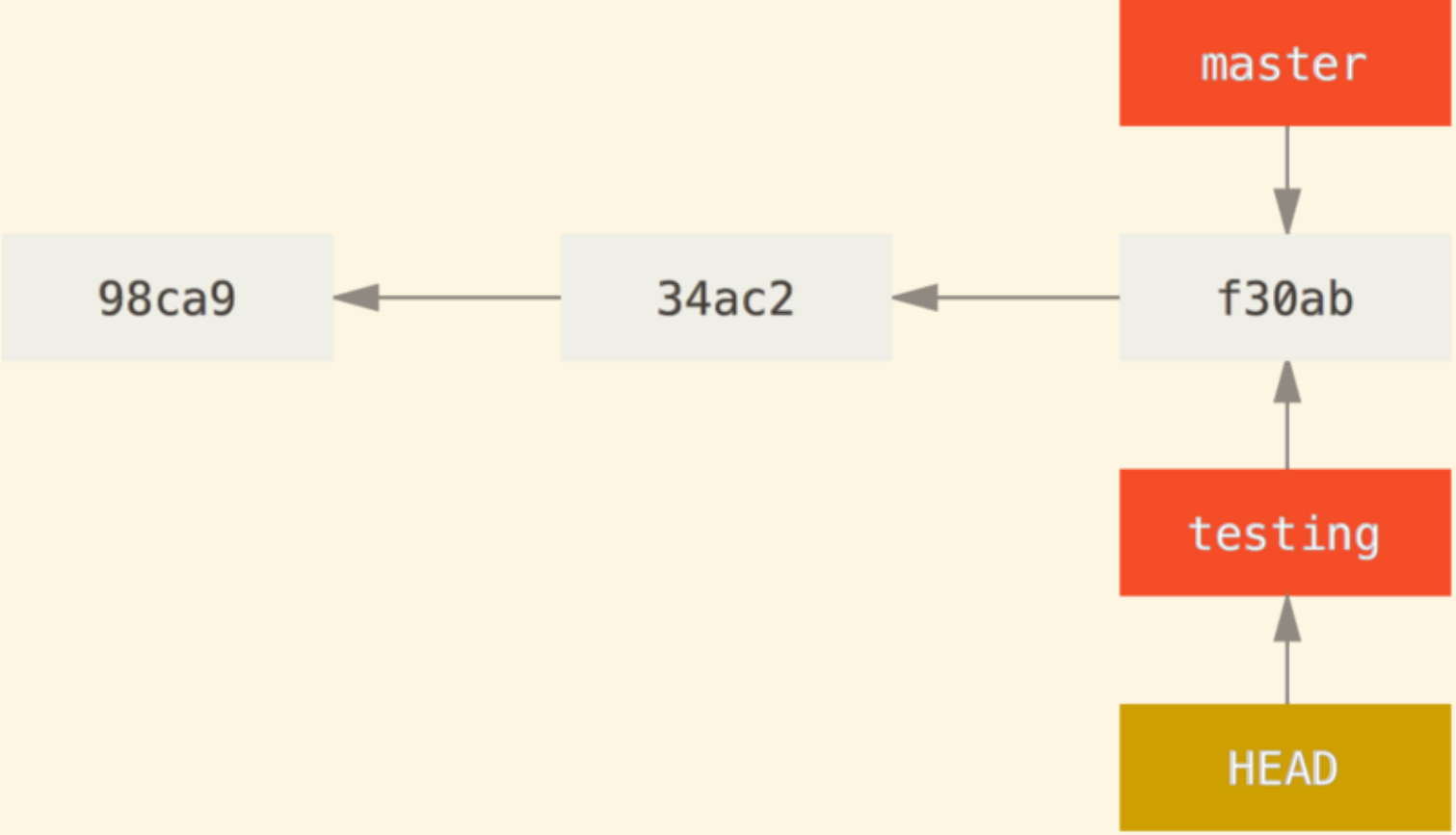


Figure 3-6. HEAD points to the current branc.

- What is the significance of that? Well, lets do another commit:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

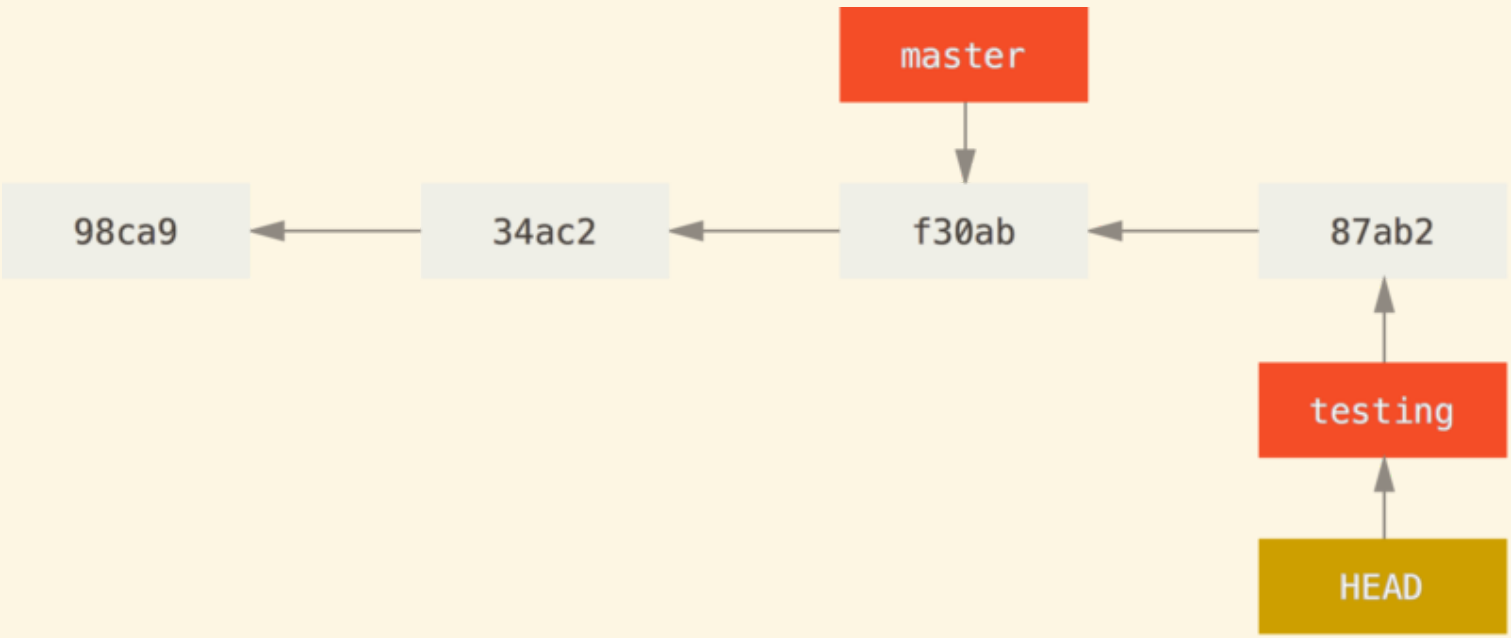


Figure 3-7. The HEAD branch moves forward when a commit is mad.

- This is interesting, because now your testing branch has moved forward, but your master branch still points to the commit you were on when you ran `git checkout` to switch branches.
- Lets switch back to the master branch:

```
$ git checkout master
```

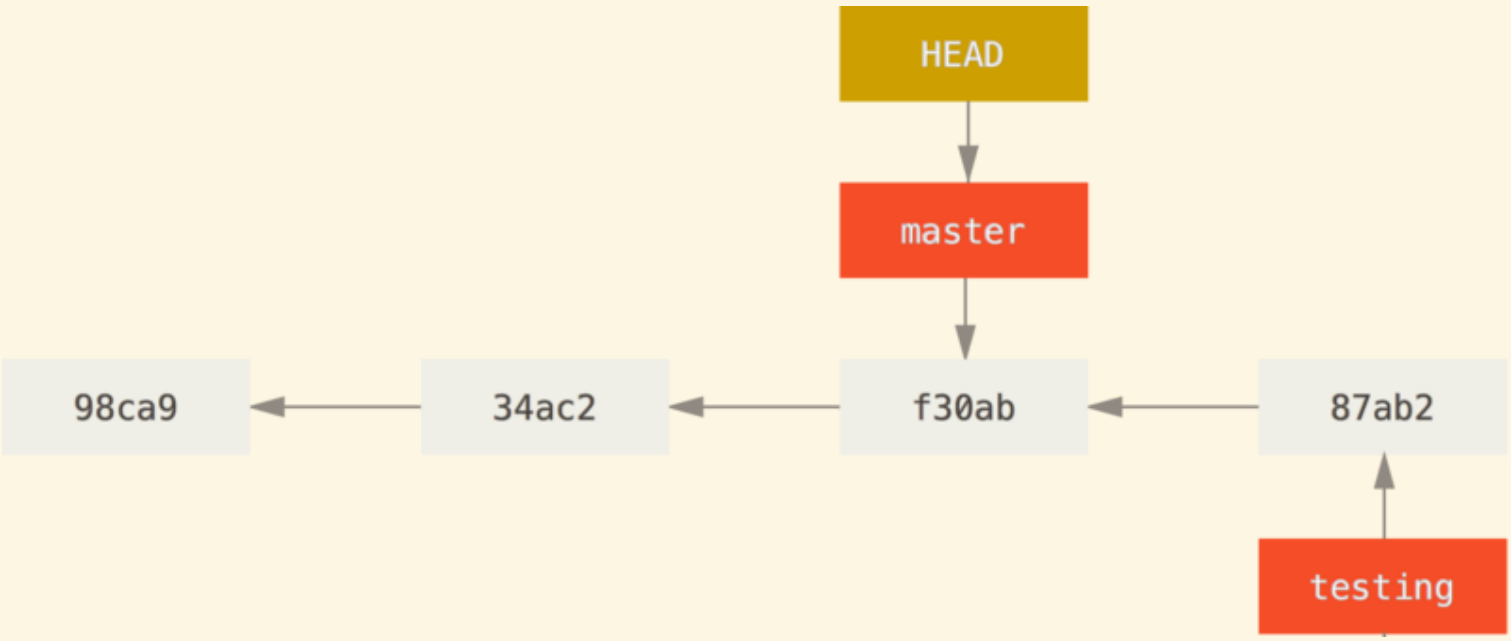


Figure 3-8. HEAD moves when you checkou.

- That command did two things.
- It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to.
- Lets make a few changes and commit again:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

- You can switch back and forth between the branches and merge them together when youre ready. And you did all that with simple `branch`, `checkout`, and `commit` commands.

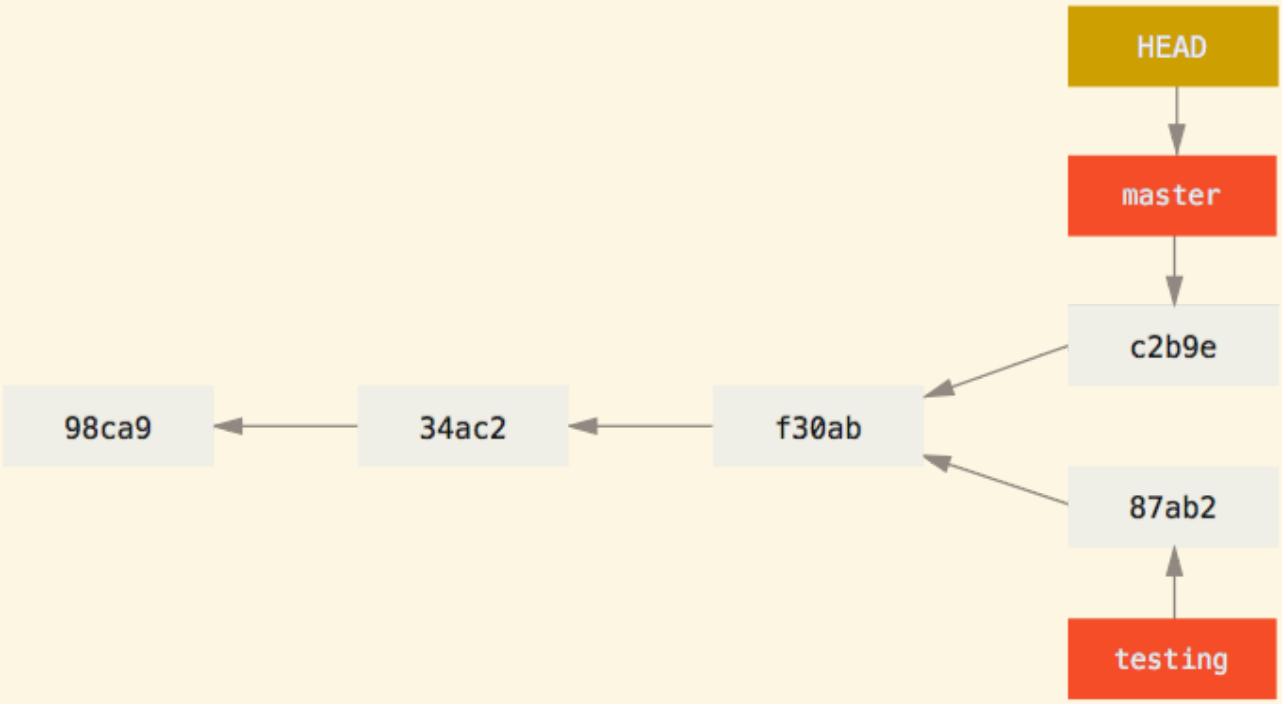


Figure 3-9. Divergent histor.

- If you run `git log --oneline --decorate --graph --all` it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Basic Branching and Merging

- You’ ll follow these steps:
    1. Do work on a web site.
    2. Create a branch for a new story youre working on.
    3. Do some work in that branch.
- At this stage, youll receive a call that another issue is critical and you need a hotfix. Youll do the following:
    1. Switch to your production branch.

2. Create a branch to add the hotfix.
3. After its tested, merge the hotfix branch, and push to production.
4. Switch back to your original story and continue working.

Basic Branching

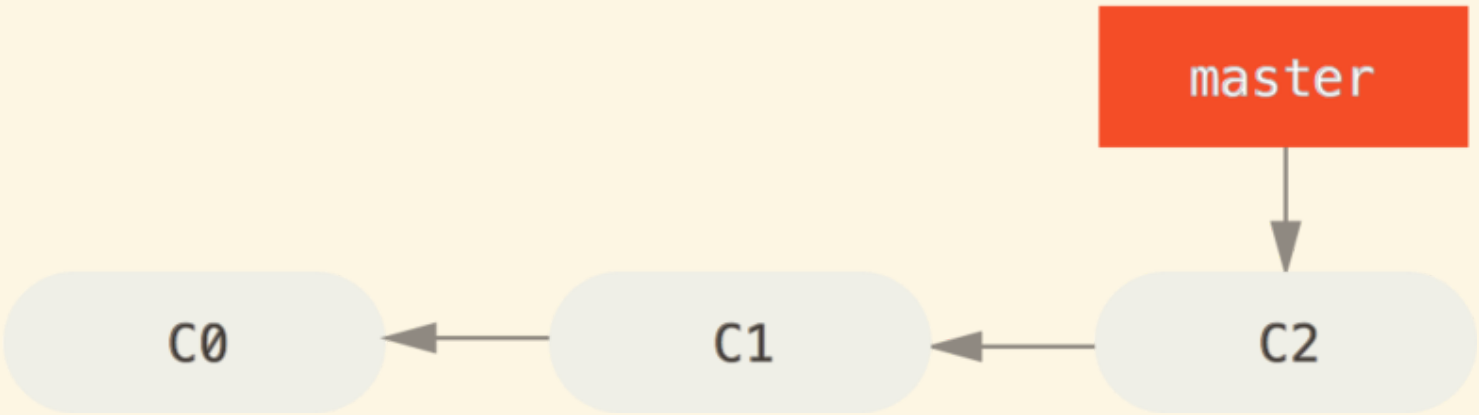


Figure 3-10. A simple commit histor.

- To create a branch and switch to it at the same time, you can run the `git checkout` command with the `-b` switch:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

- This is shorthand for:

```
$ git branch iss53
$ git checkout iss53
```

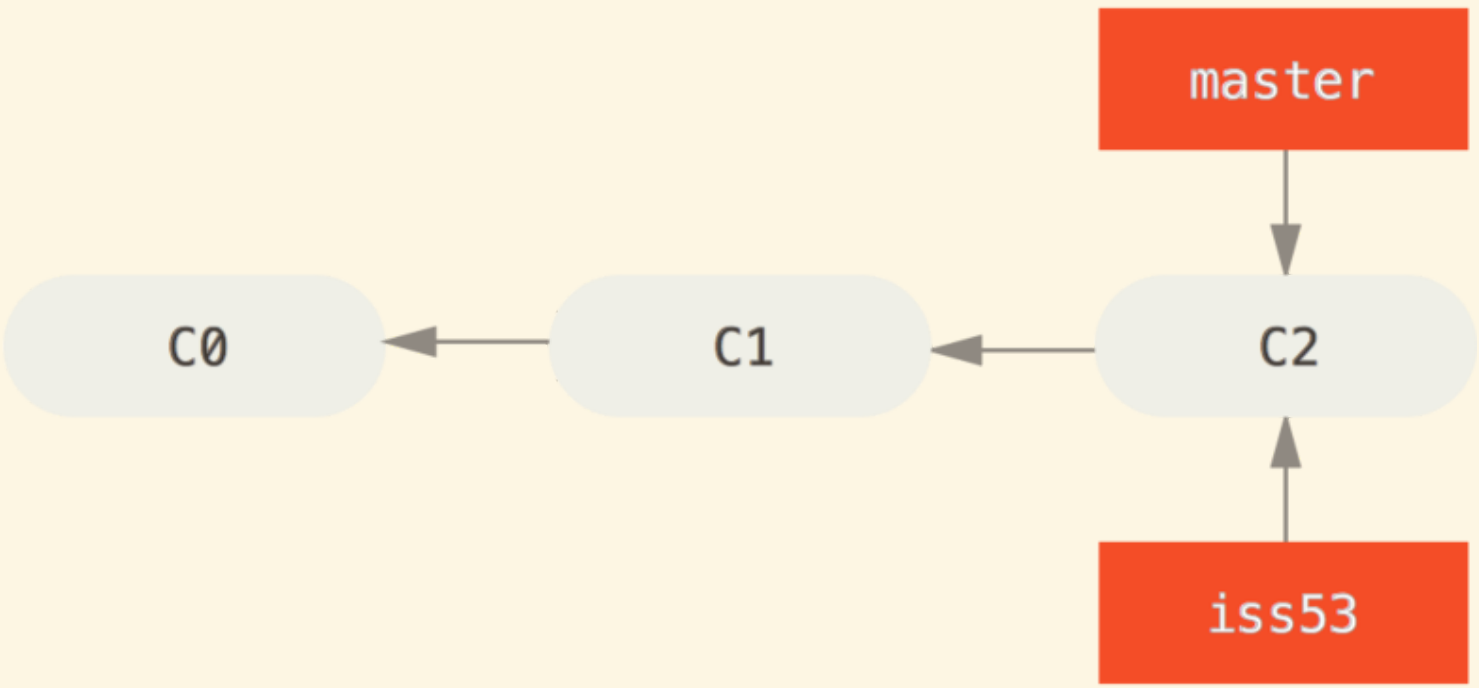


Figure 3-11. Creating a new branch pointe.

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

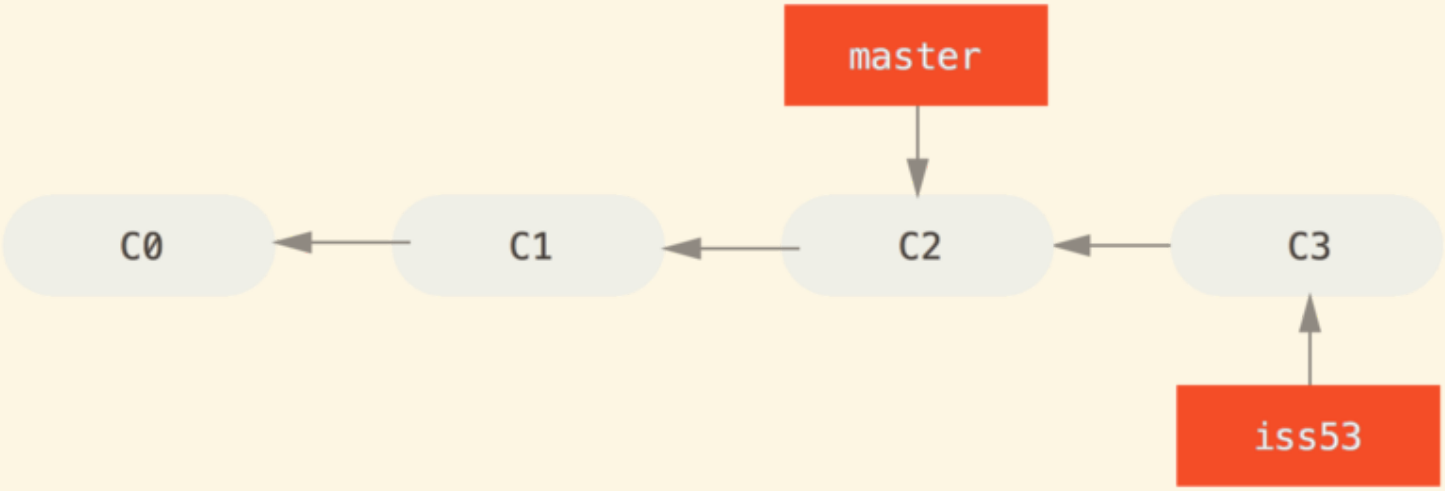


Figure 3-12. The iss53 branch has moved forward with your wor.

- Its best to have a clean working state when you switch branches.

```
$ git checkout master
Switched to branch 'master'
```

- Important point to remember: when you switch branches, Git resets your working directory to look like it did the last time you committed on that branch.

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```

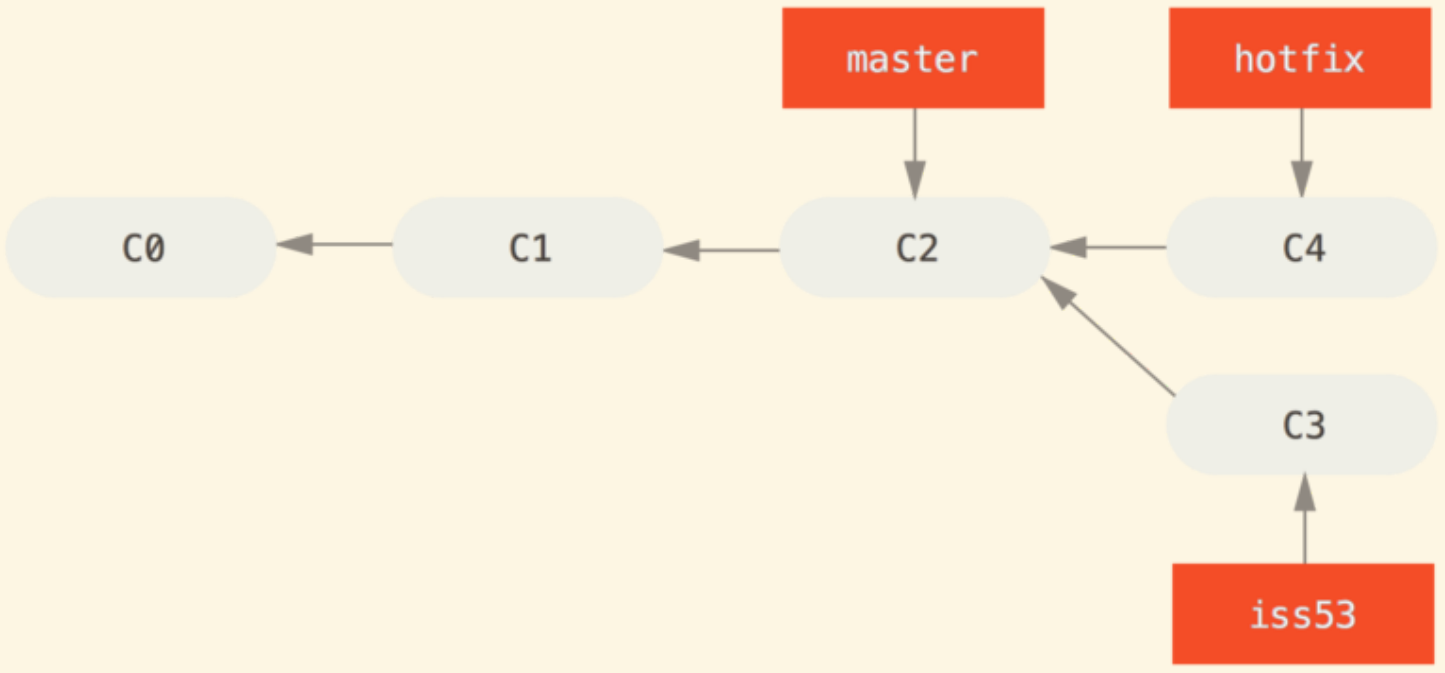


Figure 3-13. Hotfix branch based on `master.

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

- Youll notice the phrase fast-forward in that merge.
- Because the commit pointed to by the branch you merged in was directly upstream of the commit youre on, Git simply moves the pointer forward.

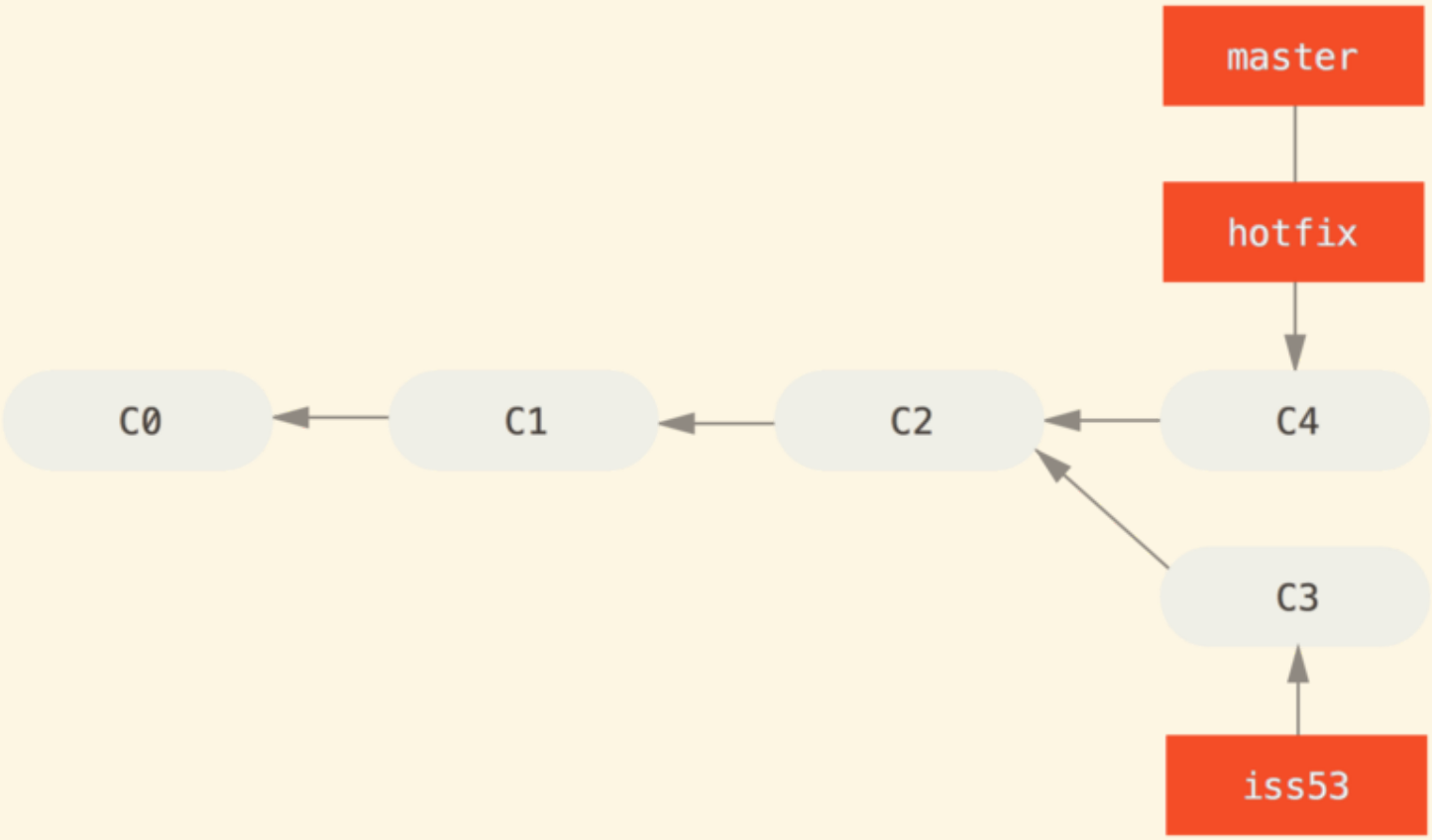


Figure 3-14. `master` is fast-forwarded to `hotfix.

- First youll delete the `hotfix` branch, because you no longer need it - the `master` branch points at the same place.
- You can delete it with the `-d` option to `git branch`:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

- Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

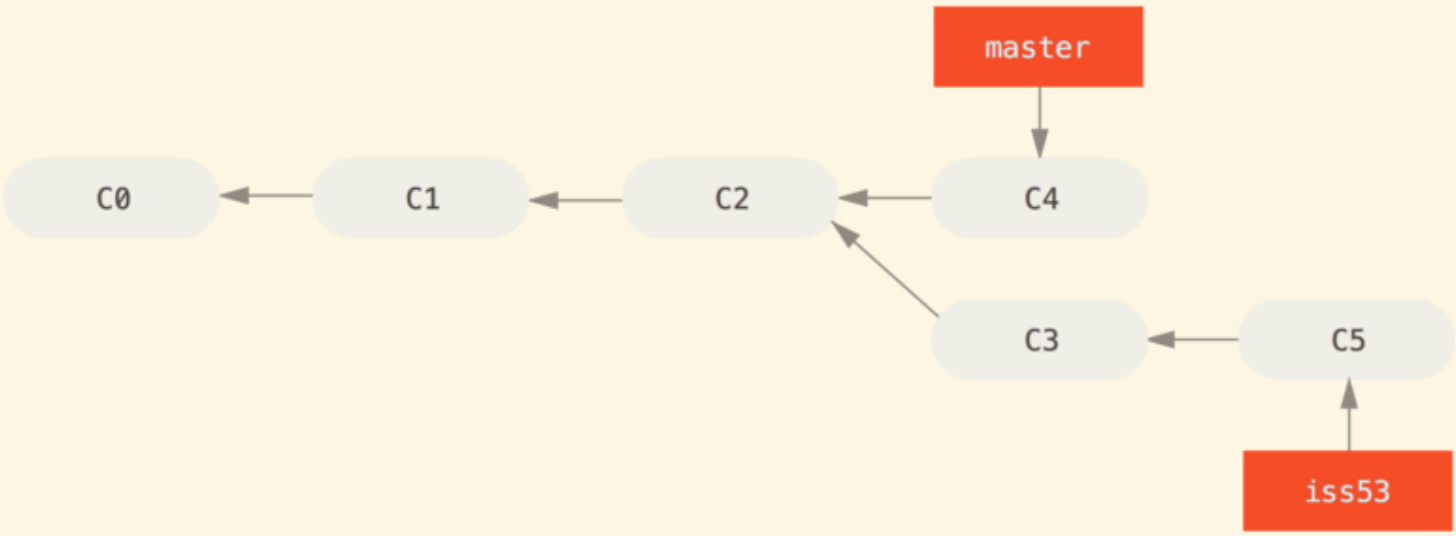


Figure 3-15. Work continues on `iss53.

- Its worth noting here that the work you did in your `hotfix` branch is not contained in the files in your `iss53` branch.
- If you need to pull it in, you can merge your `master` branch into your `iss53` branch by running `git merge master`,

Basic Merging

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

- Your development history has diverged from some older point.
- Because the commit on the branch youre on isnt a direct ancestor of the branch youre merging in, Git has to do some work.
- In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.

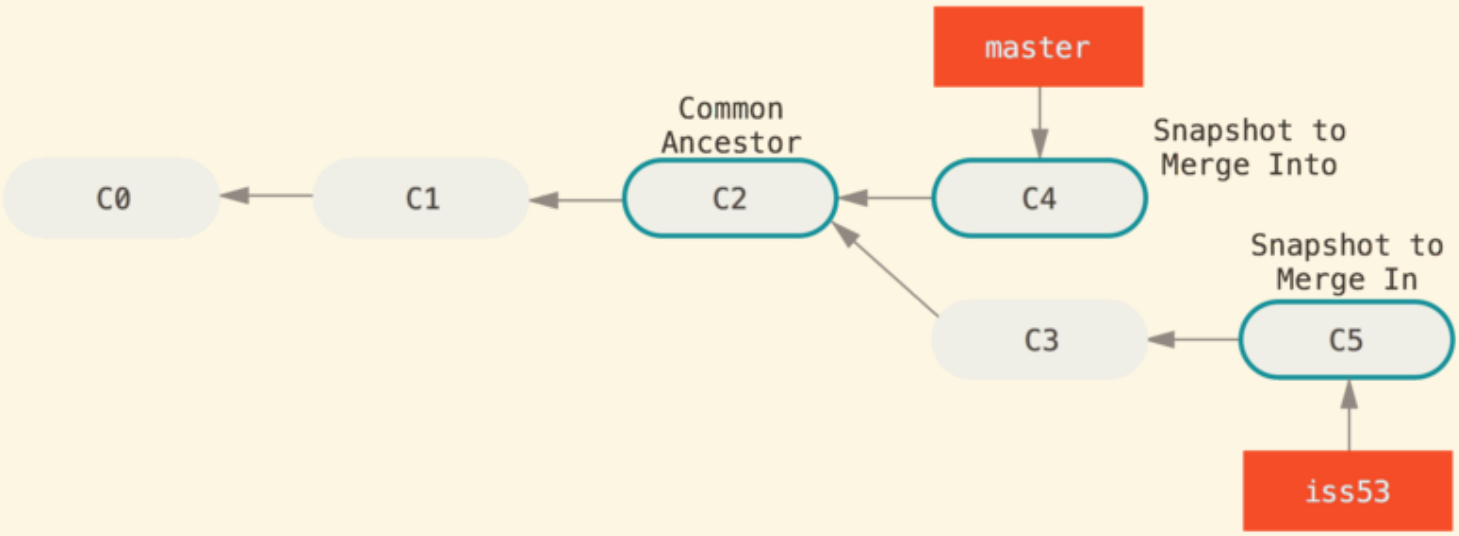


Figure 3-16. Three snapshots used in a typical merg.

- Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it.
- This is referred to as a merge commit, and is special in that it has more than one parent.

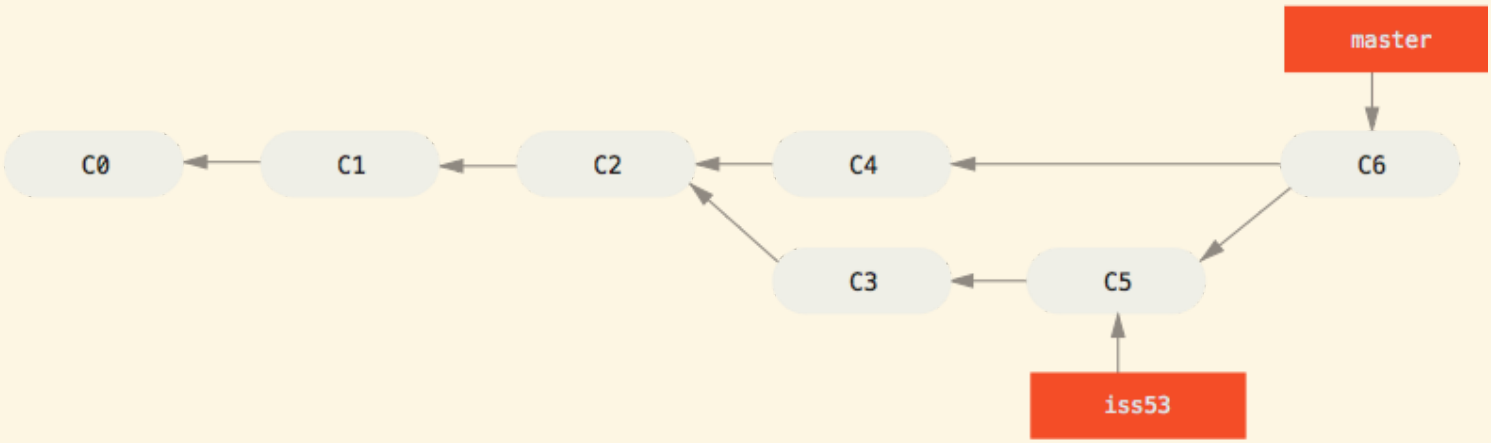


Figure 3-17. A merge commi.

- Its worth pointing out that Git determines the best common ancestor to use for its merge base; this is different than older tools like CVS or Subversion (before version 1.5), where the developer doing the merge had to figure out the best merge base for themselves.

```
$ git branch -d iss53
```

Basic Merge Conflicts

- If you changed the same part of the same file differently in the two branches youre merging together, Git wont be able to merge them cleanly.

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

- Git hasnt automatically created a new merge commit.
- It has paused the process while you resolve the conflict.

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

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

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

- After youve resolved each of these sections in each conflicted file, run `git add` on each file to mark it as resolved.
- Staging the file marks it as resolved in Git.
- If you want to use a graphical tool to resolve these issues, you can run `git mergetool`, which fires up an appropriate visual merge tool and walks you through the conflicts:

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

- Git asks you if the merge was successful.
- If you tell the script that it was, it stages the file to mark it as resolved for you.
- You can run `git status` again to verify that all conflicts have been resolved:

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#    .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#    modified:   index.html
#
```

Branch Management

- The `git branch` command does more than just create and delete branches.
- If you run it with no arguments, you get a simple listing of your current branches:

```
$ git branch
  iss53
* master
  testing
```

- Notice the `*` character that prefixes the `master` branch: it indicates the branch that you currently have checked out (i.e., the branch that `HEAD` points to).
- To see the last commit on each branch, you can run `git branch -v`:

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

- The useful `\--merged` and `\--no-merged` options can filter this list to branches that you have or have not yet merged into the branch youre currently on.
- To see which branches are already merged into the branch youre on, you can run `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

- Branches on this list without the `*` in front of them are generally fine to delete with `git branch -d`;
- To see all the branches that contain work you havent yet merged in, you can run `git branch --no-merged`:

```
$ git branch --no-merged
  testing
```

- This shows your other branch. Because it contains work that isnt merged in yet, trying to delete it with `git branch -d` will fail:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

- If you really do want to delete the branch and lose that work, you can force it with `-D`, as the helpful message points out.

Branching Workflows

Long-Running Branches

- Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do.
- In reality, were talking about pointers moving up the line of commits youre making.
- The stable branches are farther down the line in your commit history, and the bleeding-edge branches are farther up the history.

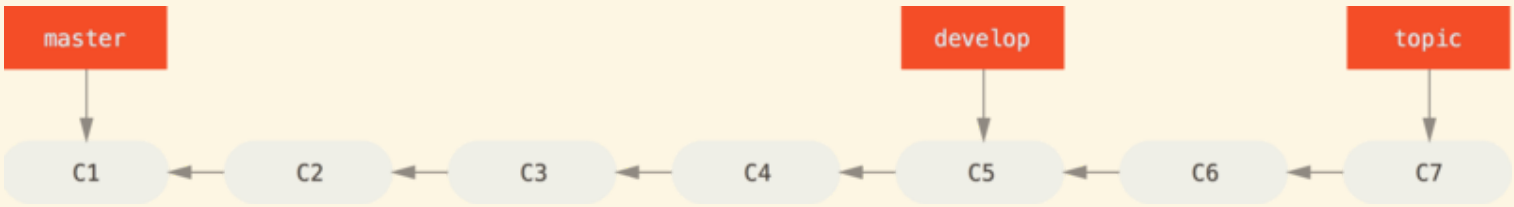


Figure 3-18. A linear view of progressive-stability branchin.

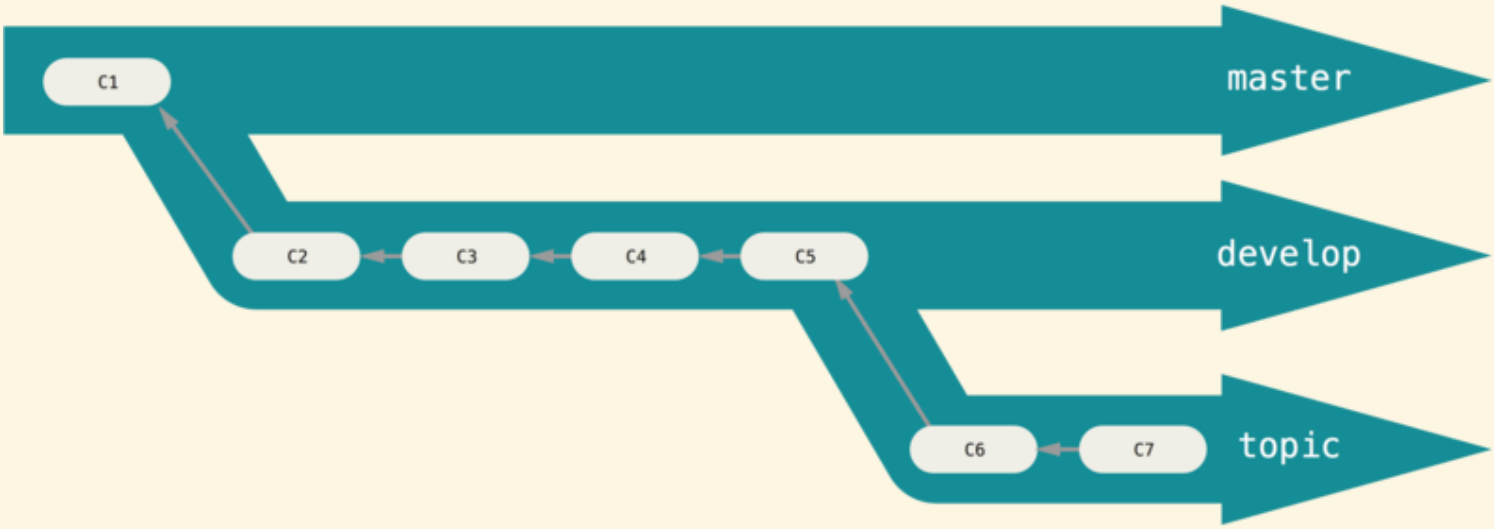


Figure 3-19. A silo view of progressive-stability branchin.

Topic Branches

- Topic branches, however, are useful in projects of any size.
- A topic branch is a short-lived branch that you create and use for a single particular feature or related work.

- Consider an example of doing some work (on `master`), branching off for an issue (`iss91`), working on it for a bit, branching off the second branch to try another way of handling the same thing (`iss91v2`), going back to your master branch and working there for a while, and then branching off there to do some work that youre not sure is a good idea (`dumbidea` branch). Your commit history will look something like this:

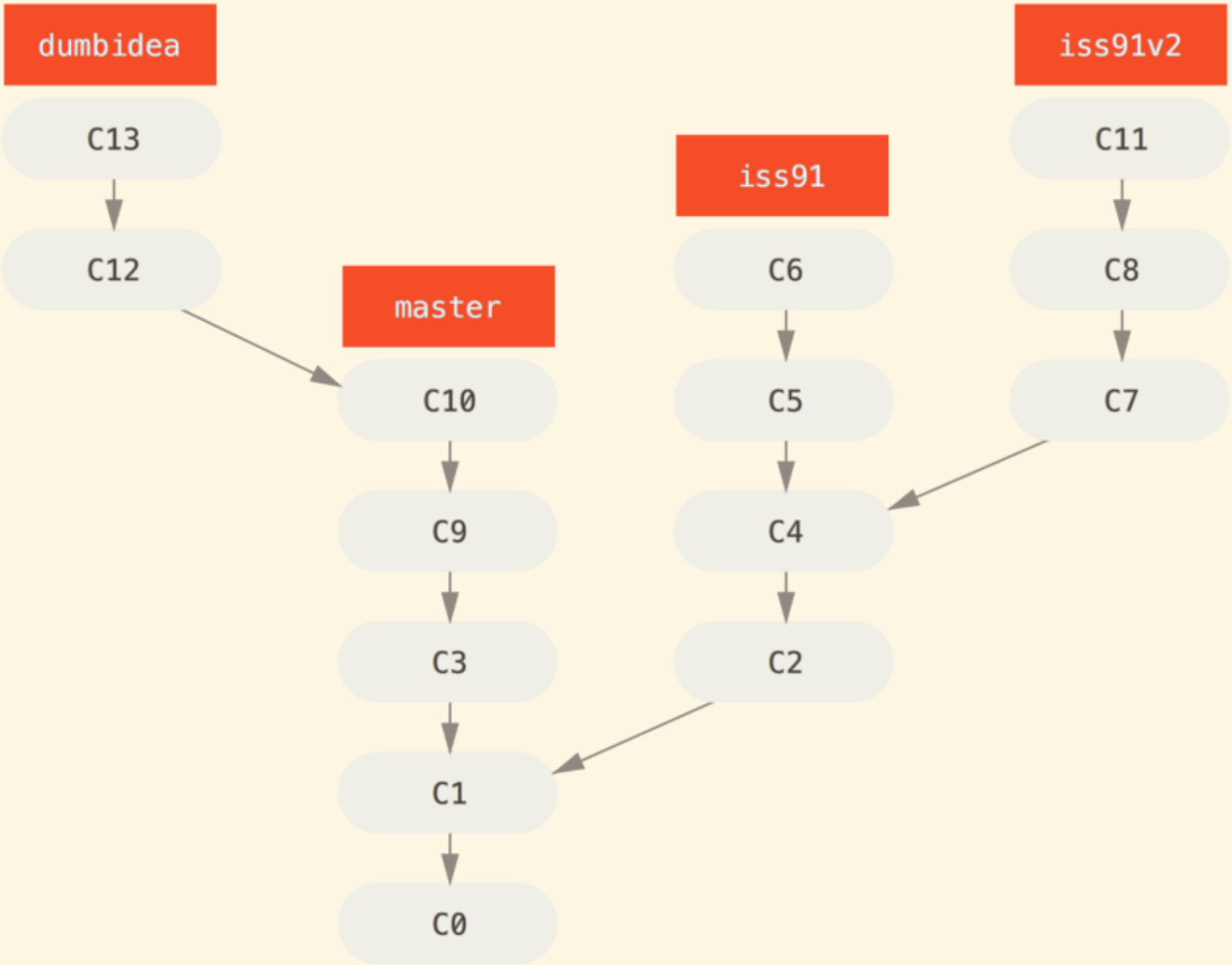


Figure 3-20. Multiple topic branche.

- Now, lets say you decide you like the second solution to your issue best (`iss91v2`); and you showed the `dumbidea` branch to your coworkers, and it turns out to be genius.
- You can throw away the original `iss91` branch (losing commits `C5` and `C6`) and merge in the other two. Your history then looks like this:

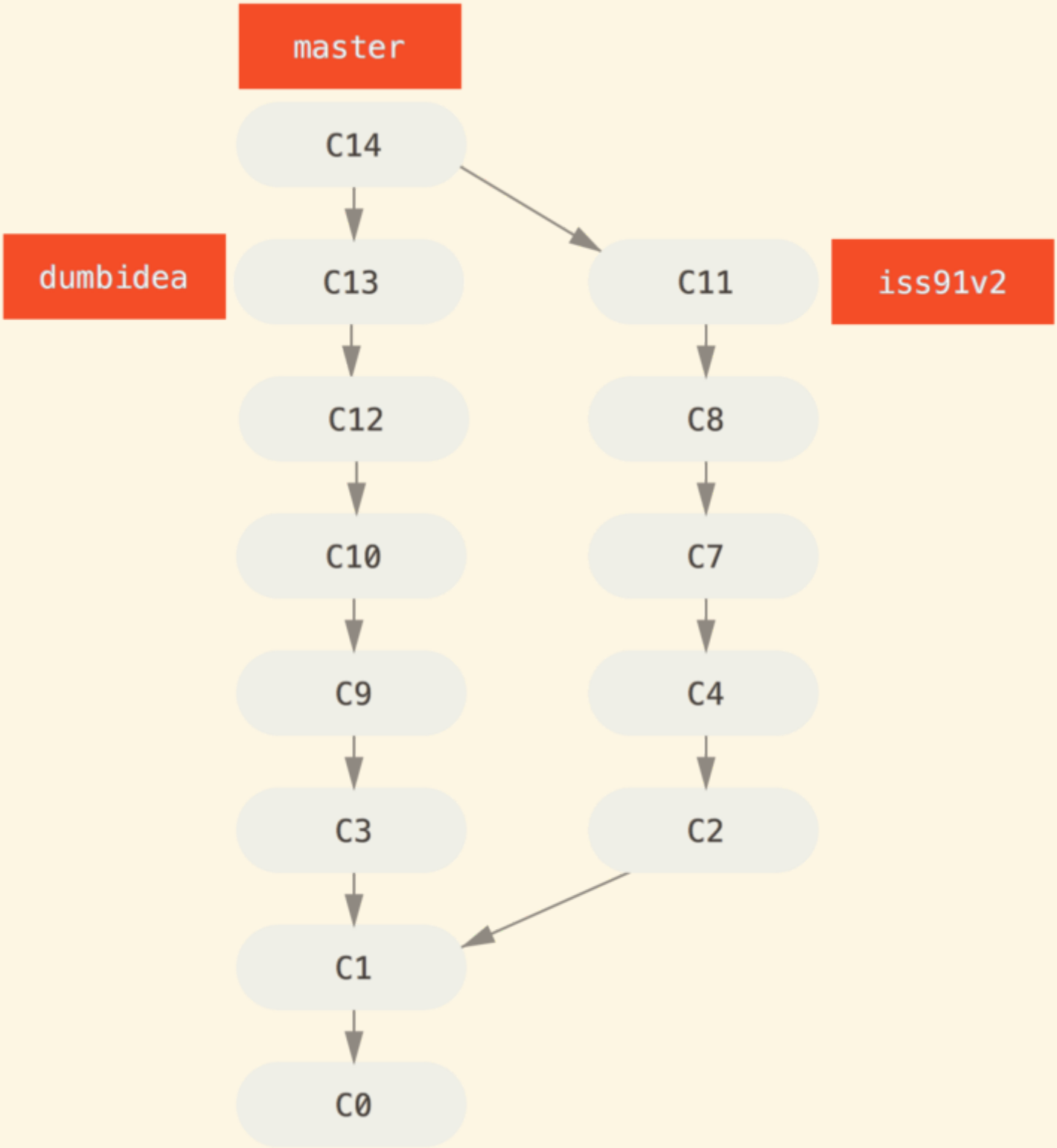


Figure 3-21. History after merging `dumbidea` and `iss91v2.

Remote Branches

- Remote branches are references (pointers) to the state of branches in your remote repositories.
- They take the form `(remote)/(branch)`.
- For instance, if you wanted to see what the `master` branch on your `origin` remote looked like as of the last time you communicated with it, you would check the `origin/master` branch.

- Lets say you have a Git server on your network at `git.ourcompany.com`.
- If you clone from this, Gits `clone` command automatically names it `origin` for you, pulls down all its data, creates a pointer to where its `master` branch is, and names it `origin/master` locally.
- Git also gives you your own local `master` branch starting at the same place as origins `master` branch, so you have something to work from.

> ***origin is not special***
> *Just like the branch name master does not have any special meaning in Git, neither does origin. While master is the default name for a starting branch when you run `git init` which is the only reason its widely used, origin is the default name for a remote when you run `git clone`. If you run `git clone -o booyah` instead, then you will have `booyah/master` as your default remote branch.*

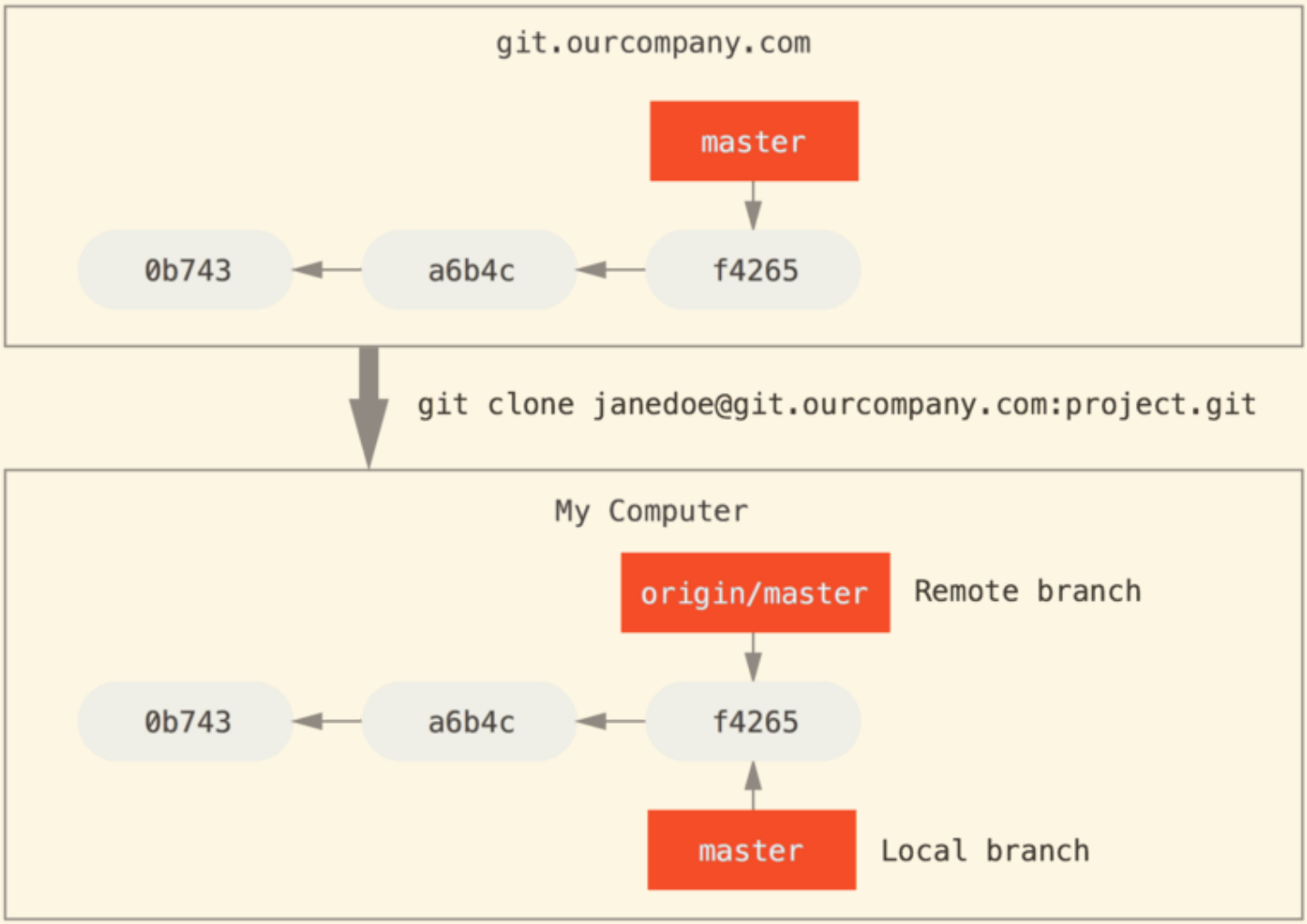


Figure 3-22. Server and local repositories after clonin.

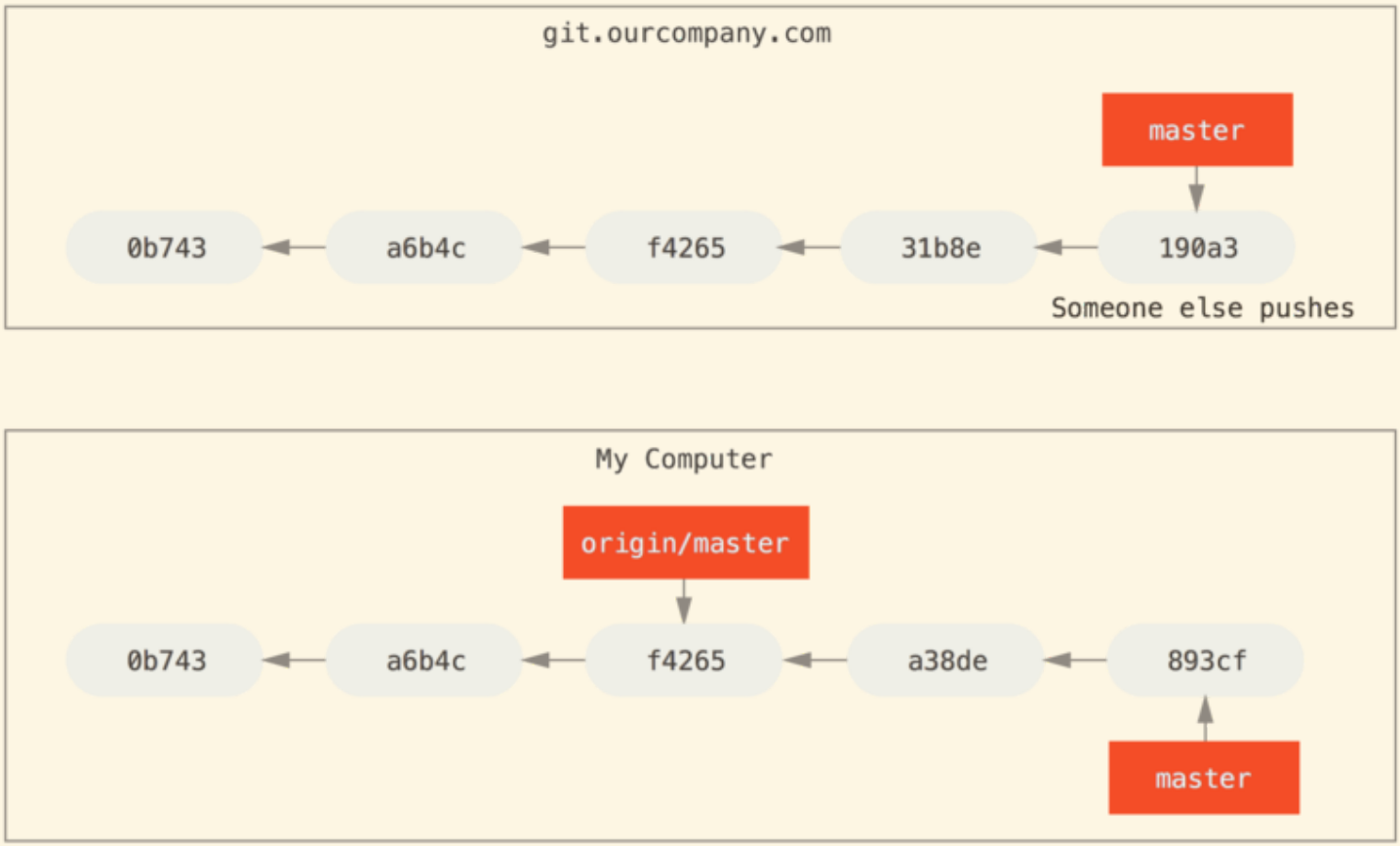


Figure 3-23. Local and remote work can diverg.

- To synchronize your work, you run a `git fetch origin` command.
- This command looks up which server origin is (in this case, its `git.ourcompany.com`), fetches any data from it that you dont yet have, and updates your local database, moving your `origin/master` pointer to its new, more up-to-date position.

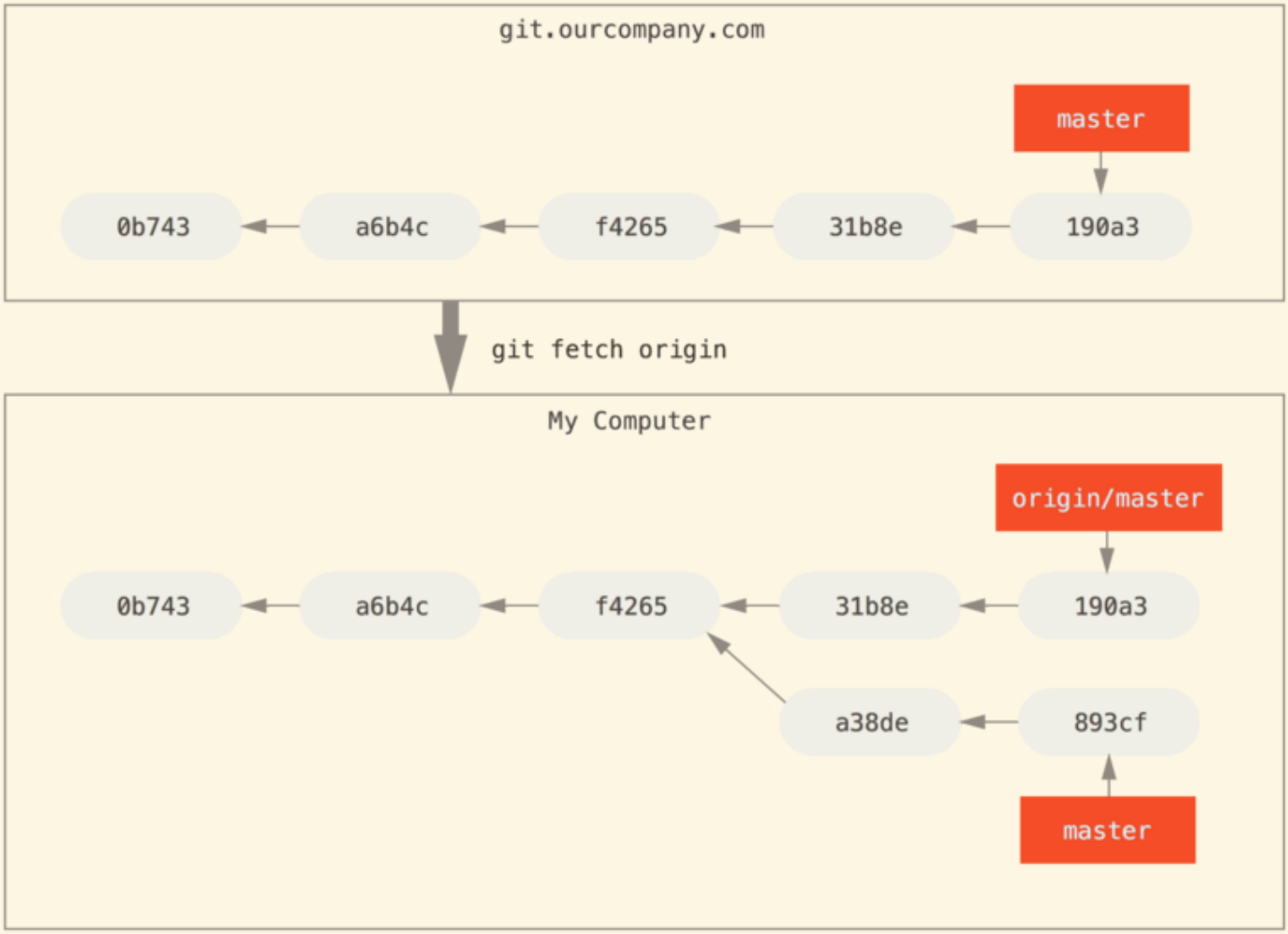


Figure 3-24. `git fetch` updates your remote reference.

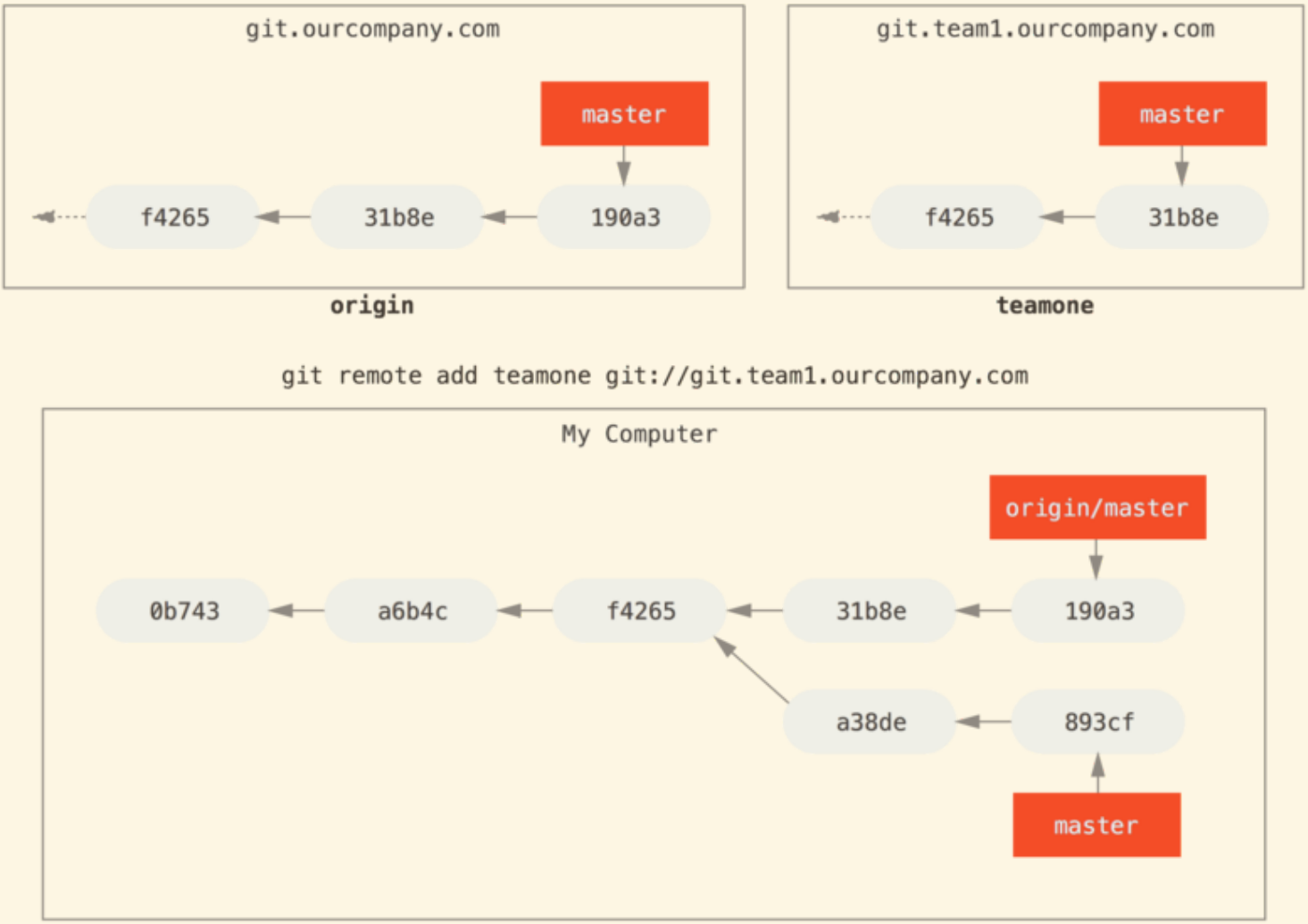


Figure 3-25. Adding another server as a remot.

- Now, you can run `git fetch teamone` to fetch everything the remote `teamone` server has that you dont have yet. Because that server has a subset of the data your `origin` server has right now, Git fetches no data but sets a remote branch called `teamone/master` to point to the commit that `teamone` has as its `master` branch.

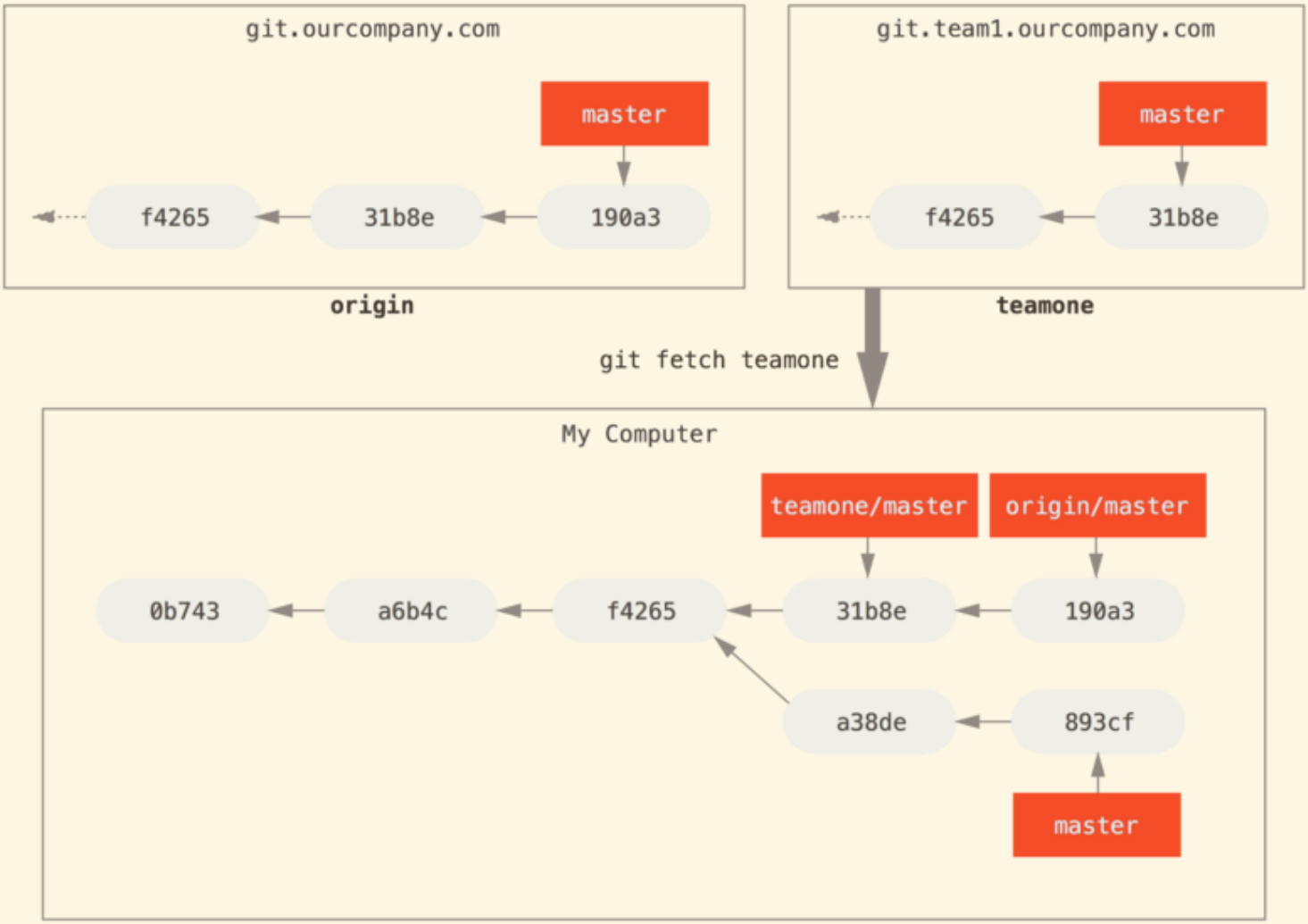


Figure 3-26. Remote tracking branch for `teamone/master.

Pushing

- Your local branches arent automatically synchronized to the remotes you write to - you have to explicitly push the branches you want to share.
- Run `git push (remote) (branch)`:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

- Git automatically expands the `serverfix` branchname out to `refs/heads/serverfix:refs/heads/serverfix`, which means, Take my serverfix local branch and push it to update the remotes serverfix branch.
- You can also do `git push origin serverfix:serverfix`, which does the same thing - it says.
- You can use this format to push a local branch into a remote branch that is named differently.

- If you didnt want it to be called `serverfix` on the remote, you could instead run `git push origin serverfix:awesomebranch` to push your local `serverfix` branch to the `awesomebranch` branch on the remote project.

> ***Dont type your password every time***
> *If youre using an HTTPS URL to push over, the Git server will ask you for your username and password for authentication. By default it will prompt you on the terminal for this information so the server can tell if youre allowed to push.*
> *If you dont want to type it every single time you push, you can set up a credential cache. The simplest is just to keep it in memory for a few minutes, which you can easily set up by running* `git config --global credential.helper cache`.

- The next time one of your collaborators fetches from the server, they will get a reference to where the servers version of `serverfix` is under the remote branch `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

- Its important to note that when you do a fetch that brings down new remote branches, you dont automatically have local, editable copies of them.
- You dont have a new `serverfix` branch - you only have an `origin/serverfix` pointer that you cant modify.
- Merge this work into your current working branch, you can run `git merge origin/serverfix`.
- If you want your own `serverfix` branch that you can work on, you can base it off your remote branch:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Tracking Branches

- Checking out a local branch from a remote branch automatically creates what is called a tracking branch (or sometimes an upstream branch).
- Tracking branches are local branches that have a direct relationship to a remote branch.
- If youre on a tracking branch and type `git pull`, Git automatically knows which server to fetch from and branch to merge into.
- Running `git checkout -b [branch] [remotename]/[branch]`.
- This is a common enough operation that git provides the `\--track` shorthand:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

- To set up a local branch with a different name than the remote branch, you can easily use the first version with a different local branch name:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

- You can use the `-u` or `\--set-upstream-to` option to `git branch` to explicitly set it at any time.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

> *Upstream shorthand*
> *When you have a tracking branch set up, you can reference it with the `@{upstream}` or `@{u}` shorthand. So if youre on the `master` branch and its tracking `origin/master`, you can say something like `git merge @{u}` instead of `git merge origin/master` if you wish.*

- If you want to see what tracking branches you have set up, you can use the `-vv` option to `git branch`.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

- Its important to note that these numbers are only since the last time you fetched from each server.
- This command does not reach out to the servers, its telling you about what it has cached from these servers locally.
- If you want totally up to date ahead and behind numbers, youll need to fetch from all your remotes right before running this.
- You could do that like this: `$ git fetch --all; git branch -vv`

Pulling

- While the `git fetch` command will fetch down all the changes on the server that you dont have yet, it will not modify your working directory at all.
- It will simply get the data for you and let you merge it yourself.
- However, there is a command called `git pull` which is essentially a `git fetch` immediately followed by a `git merge` in most cases.

- `git pull` will look up what server and branch your current branch is tracking, fetch from that server and then try to merge in that remote branch.

Deleting Remote Branches

- You can delete a remote branch using the `\\--delete` option to `git push`.
- If you want to delete your `serverfix` branch from the server, you run the following:

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

Rebasing

- In Git, there are two main ways to integrate changes from one branch into another: the `merge` and the `rebase`.

The Basic Rebase

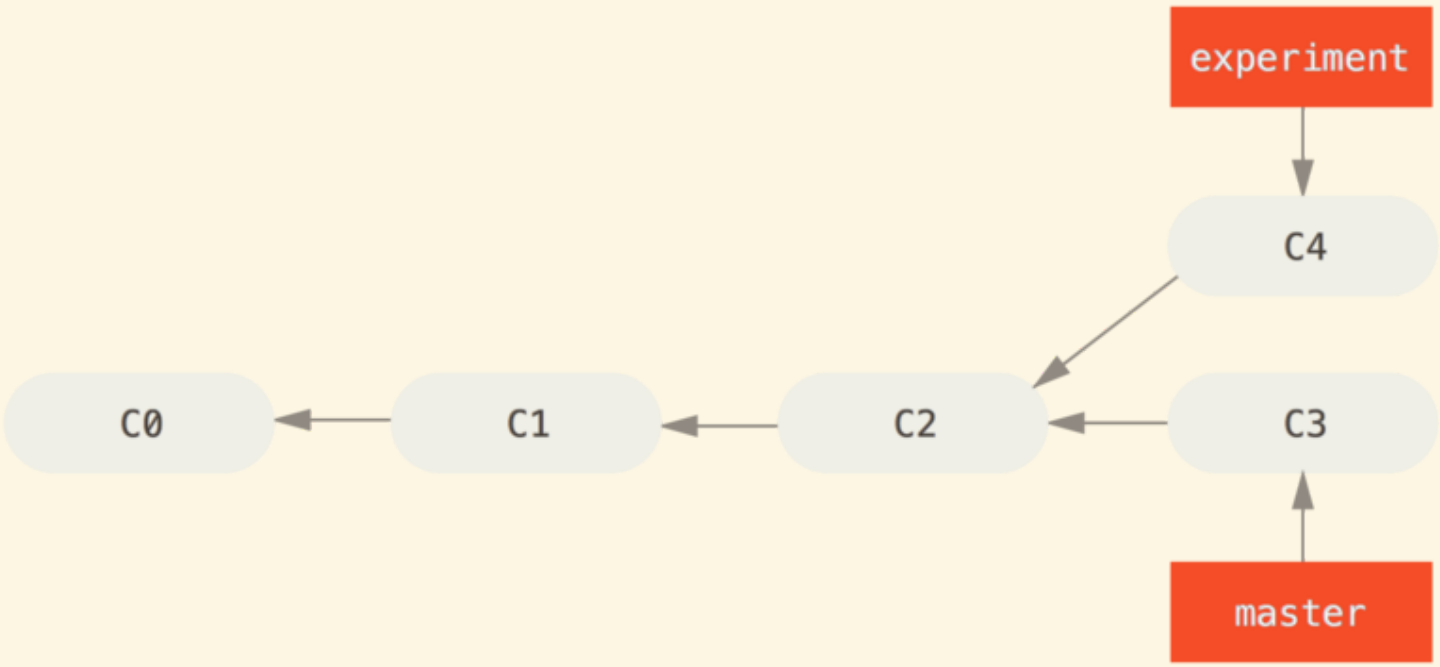


Figure 3-27. Simple divergent histor.

- 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).

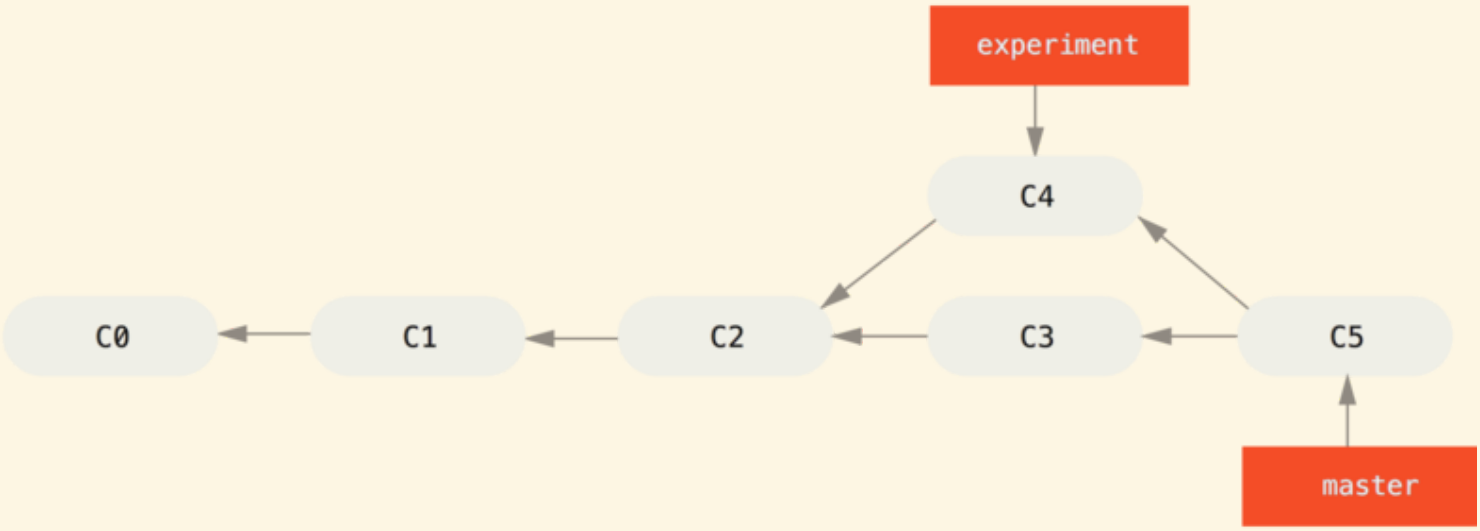


Figure 3-28. Merging to integrate diverged work histor.

- However, there is another way: you can take the patch of the change that was introduced in `C4` and reapply it on top of `C3`.
- 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, youd 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 youre on and the one youre rebasing onto), getting the diff introduced by each commit of the branch youre 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.

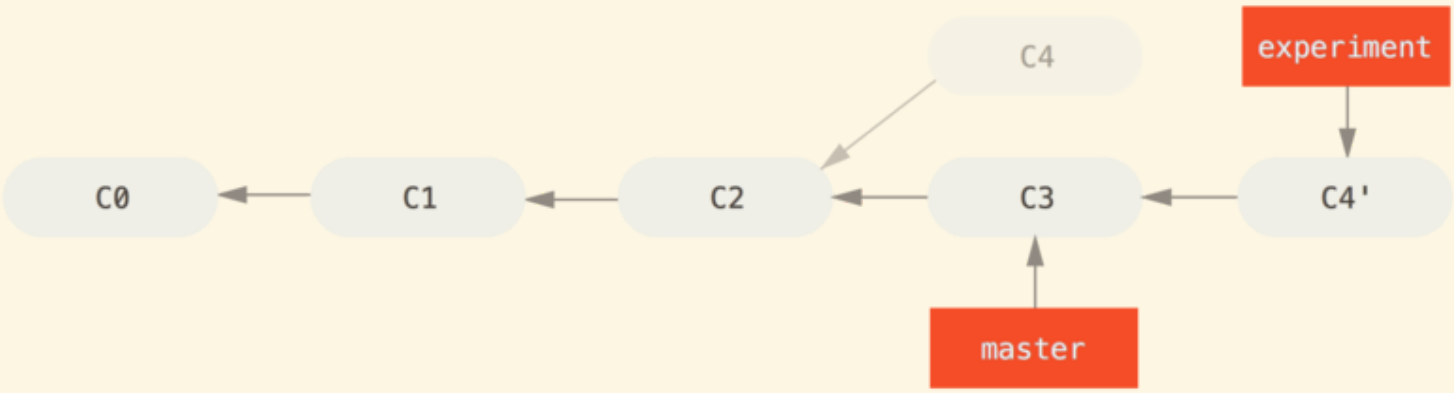


Figure 3-29. Rebasing the change introduced in `C4` onto `C3.

- At this point, you can go back to the master branch and do a fast-forward merge.

```
$ git checkout master
$ git merge experiment
```

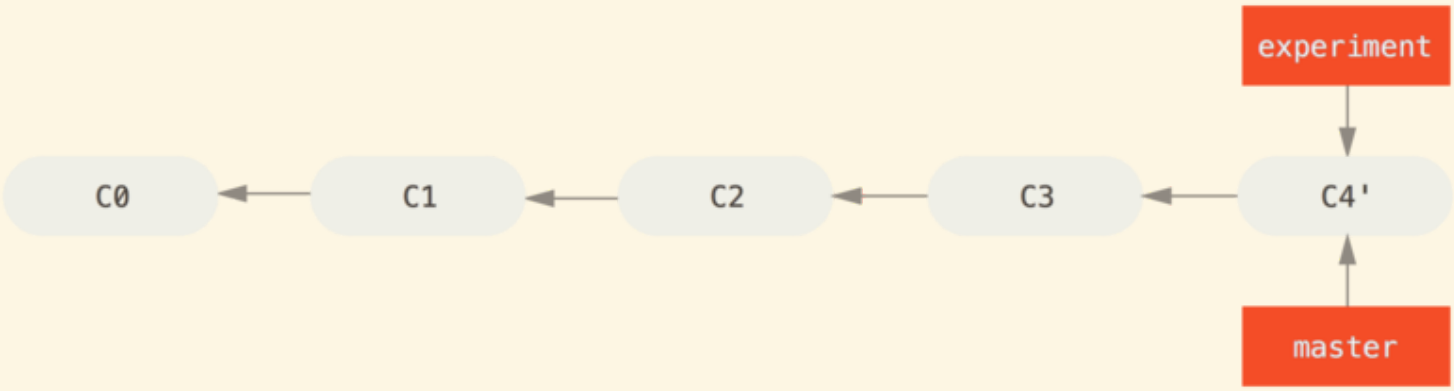


Figure 3-30. Fast-forwarding the master branc.

- 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, youll do this to make sure your commits apply cleanly on a remote branch - perhaps in a project to which youre trying to contribute but that you dont maintain.
- In this case, youd 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 doesnt have to do any integration work - just a fast-forward or a clean apply.
- 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 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.

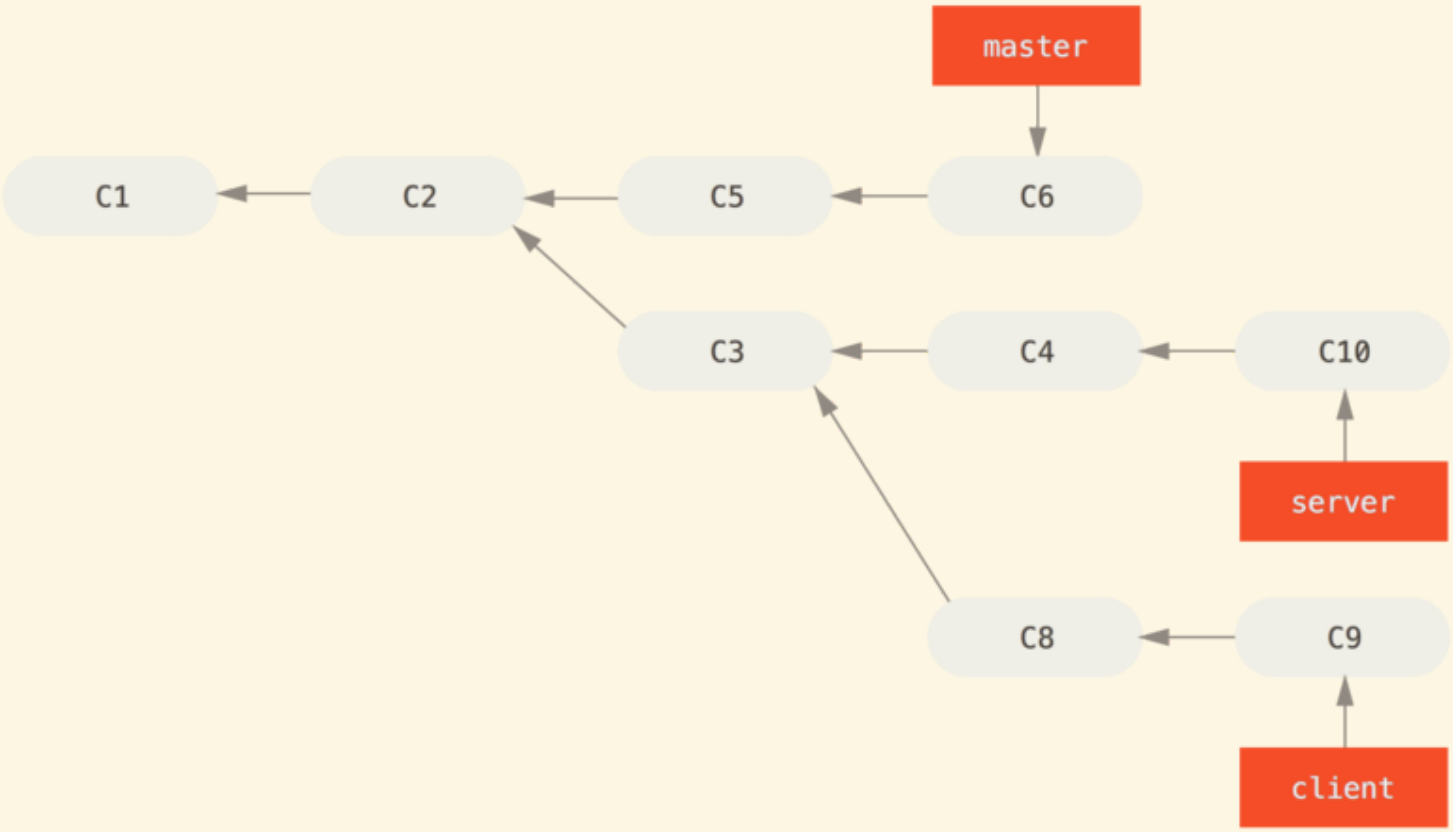


Figure 3-31. A history with a topic branch off another topic branc.

- 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 its tested further.
- You can take the changes on client that arent 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
```

- 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`.

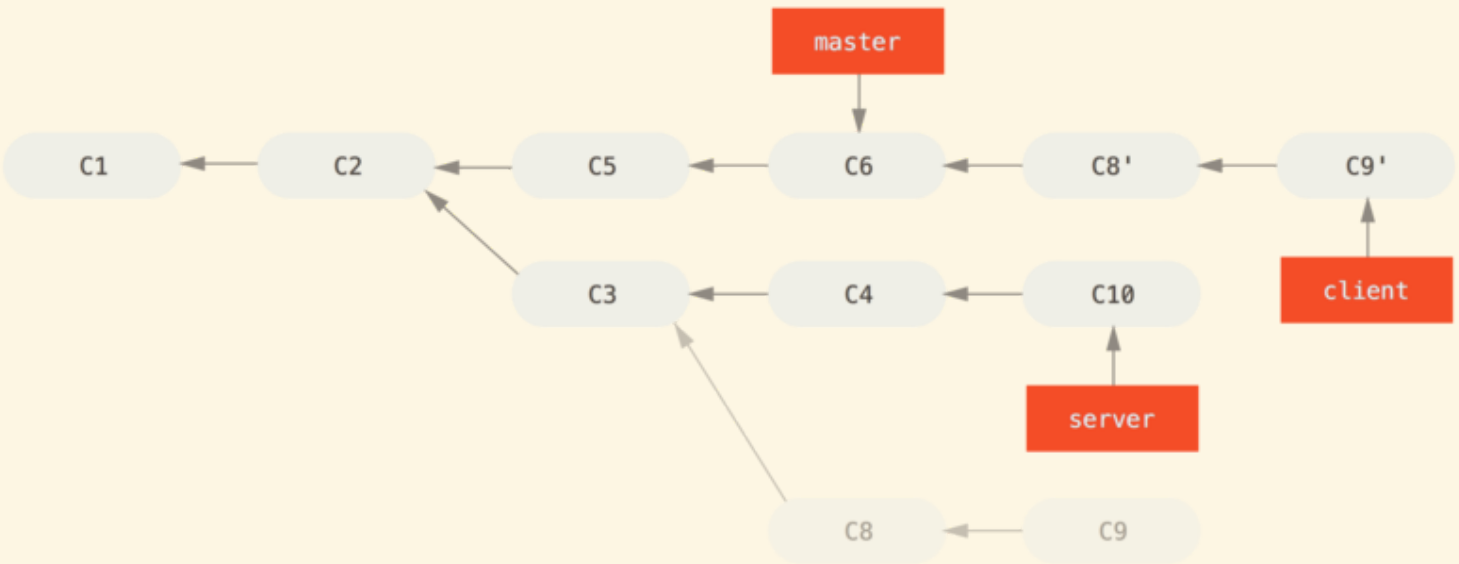


Figure 3-32. Rebasing a topic branch off another topic branc.

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

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

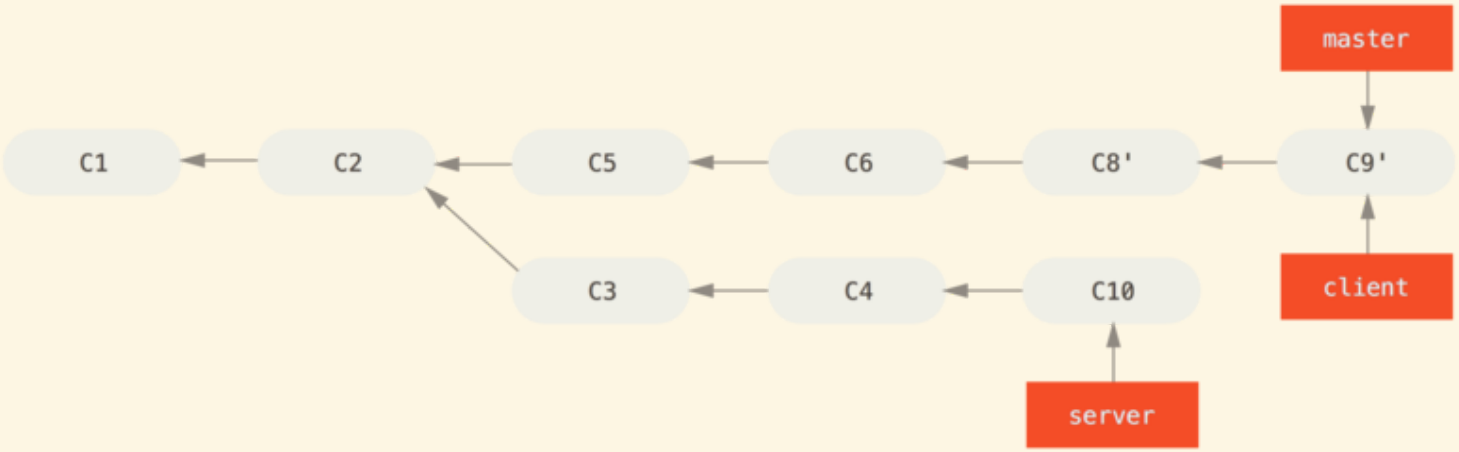
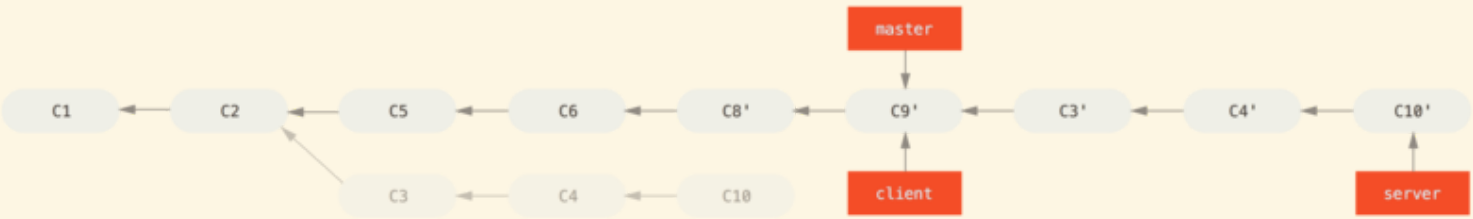


Figure 3-33. Fast-forwarding your master branch to include the clien.

Lets say you decide to pull in your server branch as well. `git rebase [basebranch] [topicbranch]` `server` `master`

`git $ 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 branc.*

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

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

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

Figure 3-35. Final commit histor.

The Perils of Rebasing

Do not rebase commits that exist outside your repository.

- Lets look at an example of how rebasing work that youve made public can cause problems.
- Suppose you clone from a central server and then do some work off that.
- Your commit history looks like this:

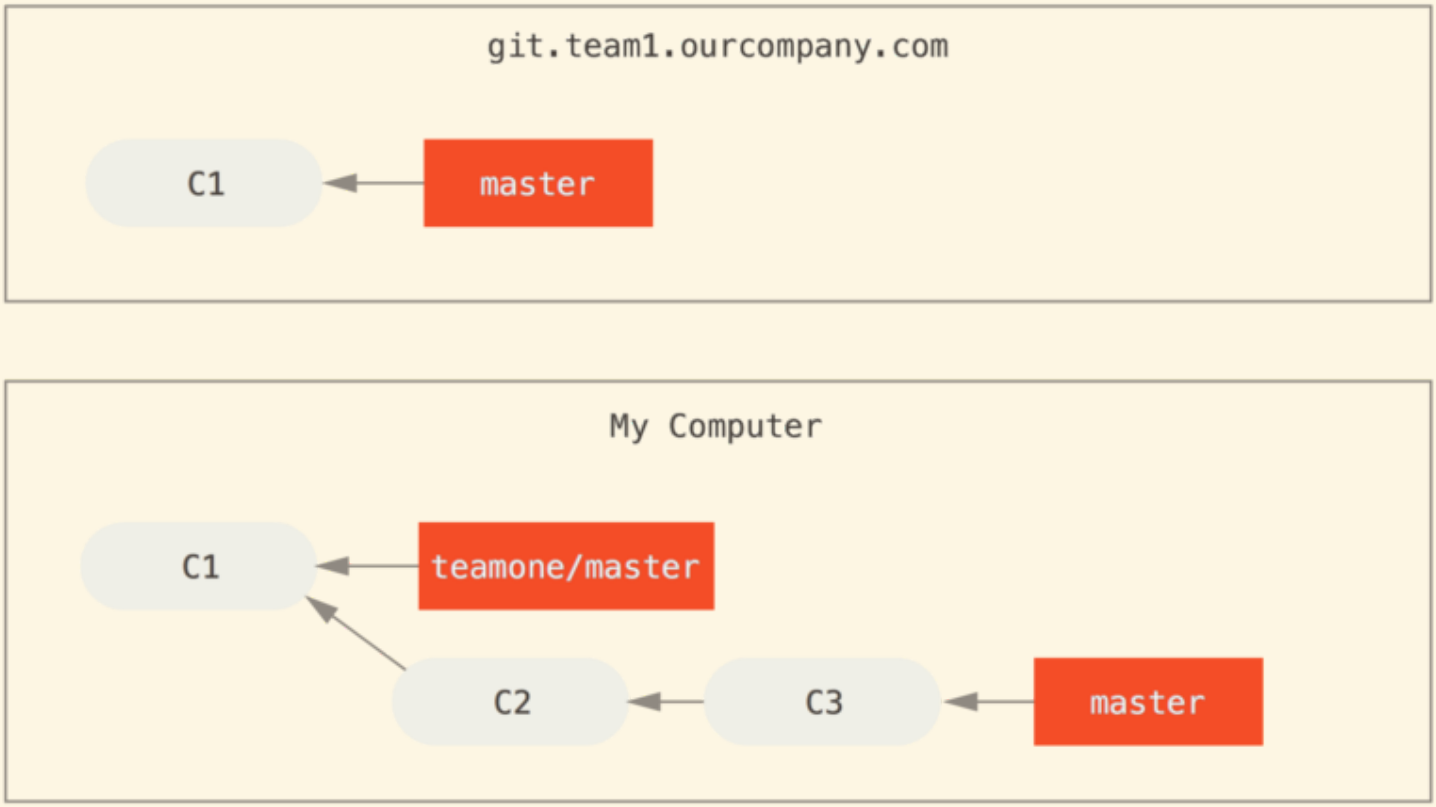


Figure 3-36. Clone a repository, and base some work on i.

- 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 this:

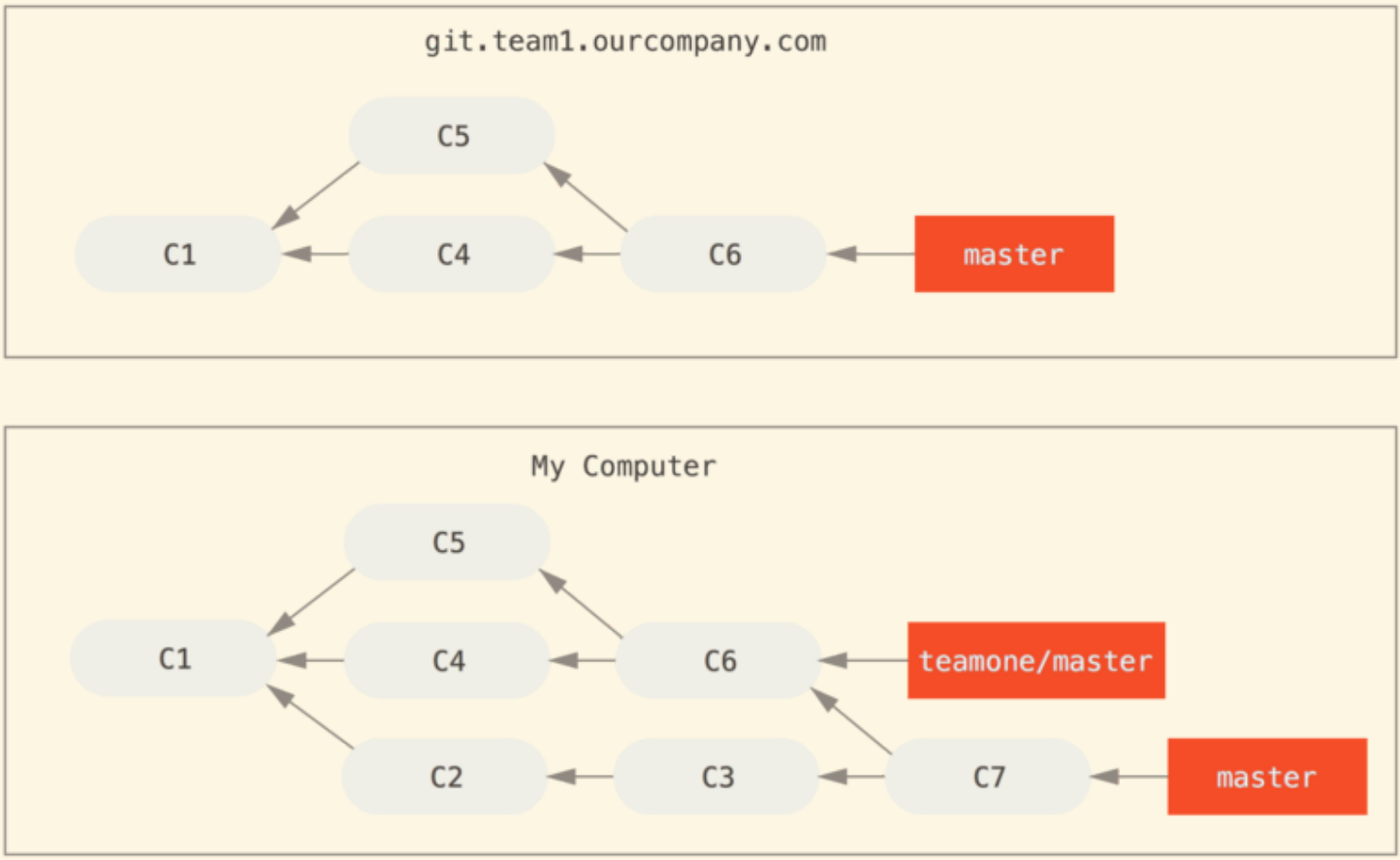


Figure 3-37. Fetch more commits, and merge them into your wor.

- 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.

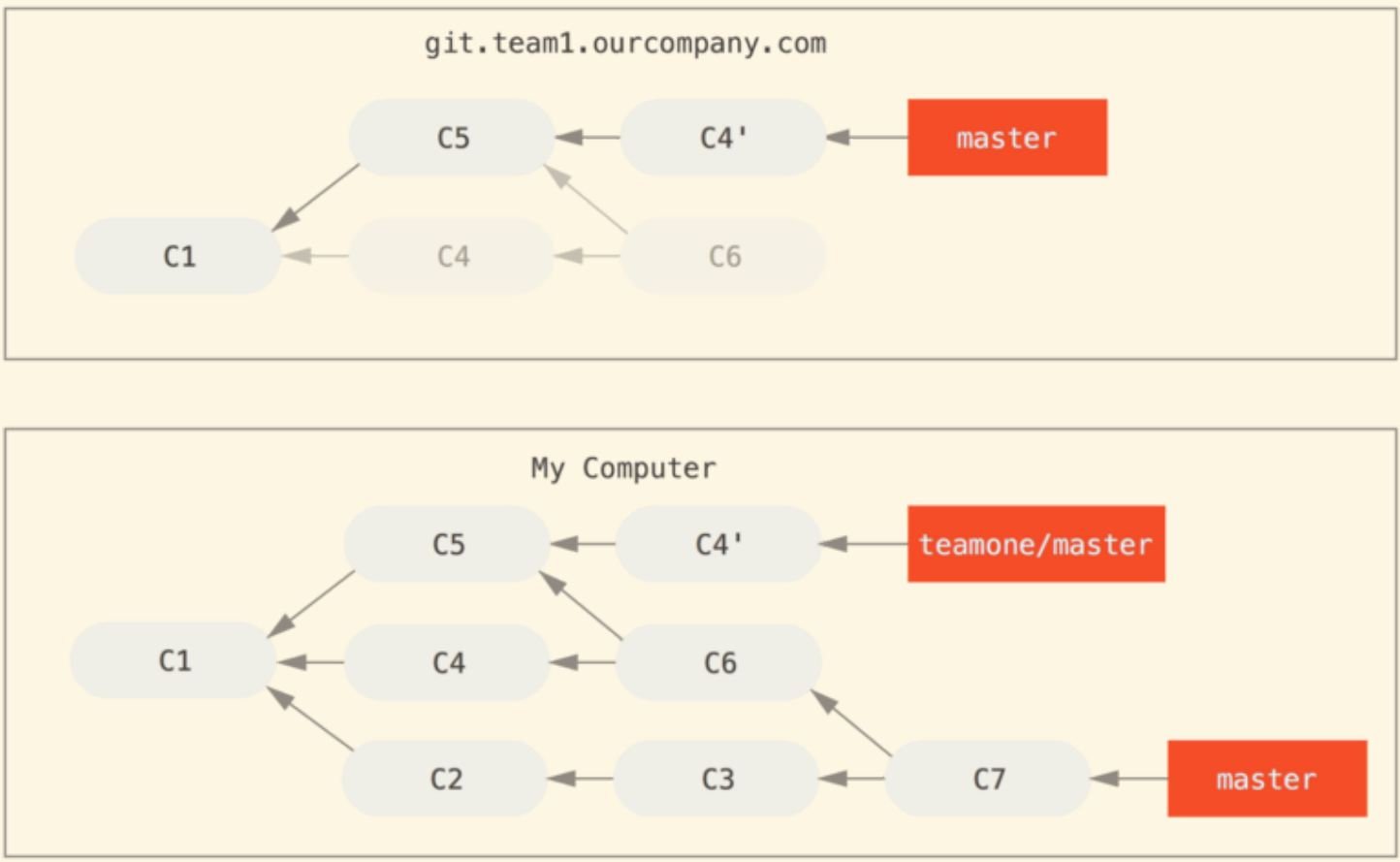
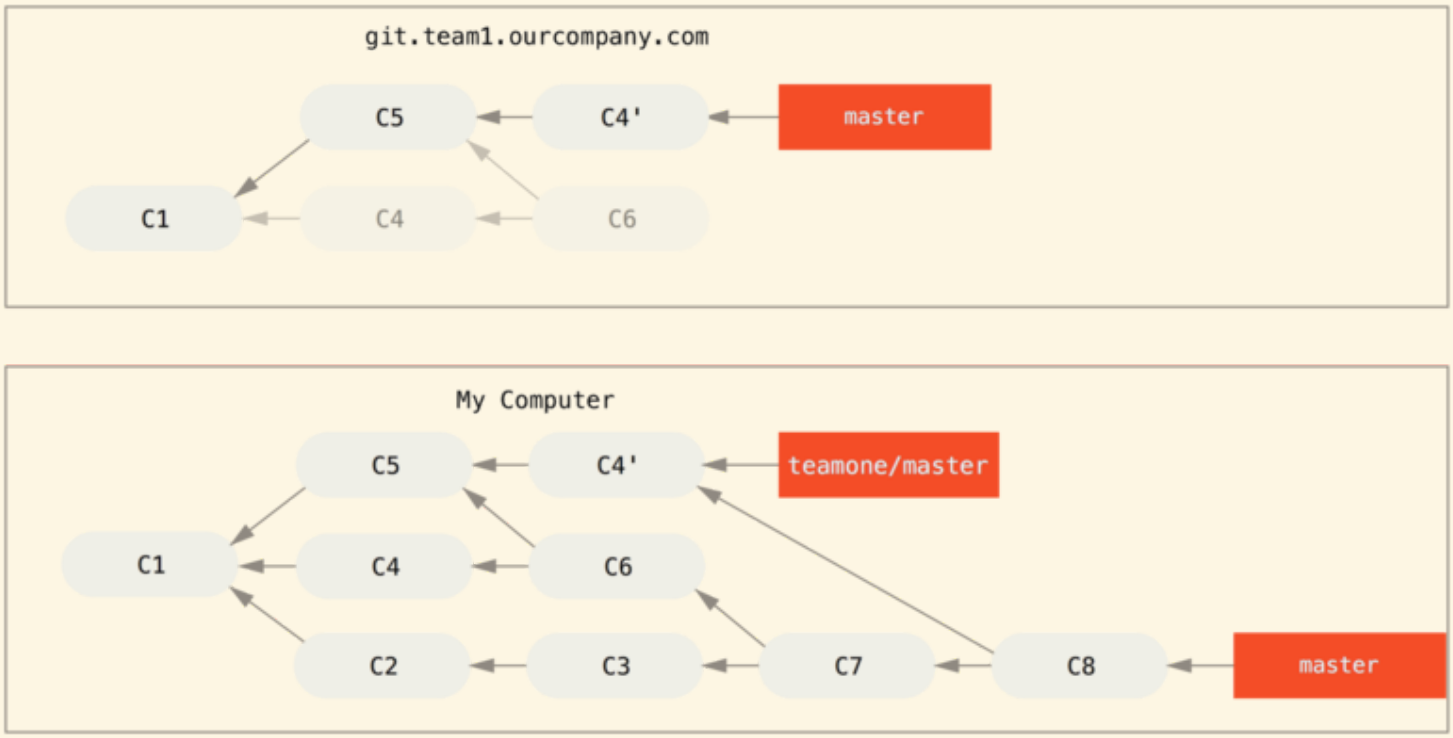


Figure 3-38. Someone pushes rebased commits, abandoning commits youv.

Now youre both in a pickle. If you do a `git pull`, youll create a merge commit which includes both lines of history, and your repository will look like this:



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

- Its pretty safe to assume that the other developer doesnt want `C4` and `C6` to be in the history; thats why she rebased in the first place.

Rebase When You Rebase

- If someone on your team force pushes changes that overwrite work that youve based work on, your challenge is to figure out what is yours and what theyve rewritten.
- It turns out that in addition to the commit SHA-1 checksum, Git also calculates a checksum that is based just on the patch introduced with the commit. This is called a patch-id.
- Determine what work is unique to our branch (C2, C3, C4, C6, C7)
- Determine which are not merge commits (C2, C3, C4)
- Determine which have not been rewritten into the target branch (just C2 and C3, since C4 is the same patch as C4')
- Apply those commits to the top of `teamone/master`

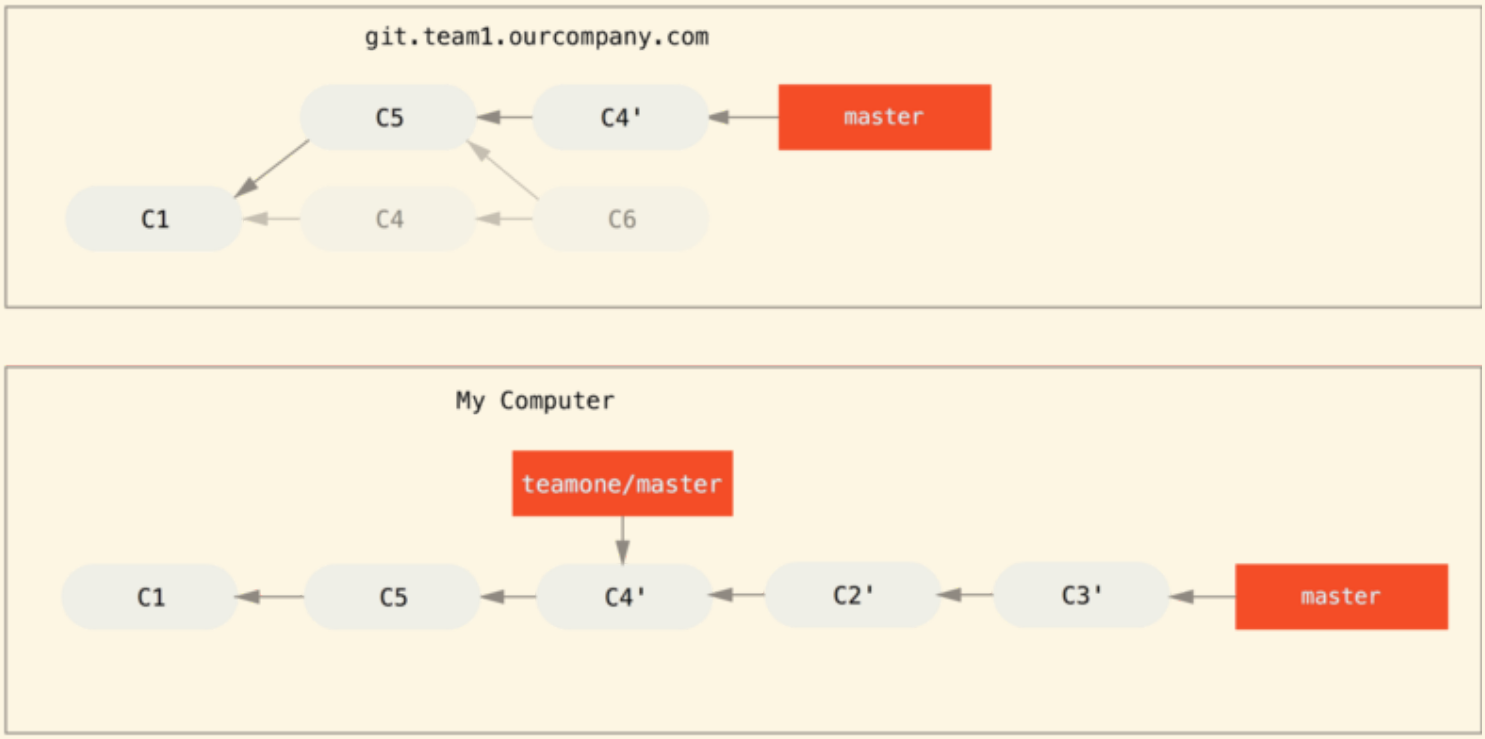


Figure 3-40. Rebase on top of force-pushed rebase work.

- You can also simplify this by running a `git pull --rebase` instead of a normal `git pull`.
- Or you could do it manually with a `git fetch` followed by a `git rebase teamone/master` in this case.
- If you are using `git pull` and want to make `\--rebase` the default, you can set the `pull.rebase` config value with something like `git config --global pull.rebase true`.
- If you or a partner does find it necessary at some point, make sure everyone knows to run `git pull --rebase` to try to make the pain after it happens a little bit simpler.

Rebase vs. Merge

- One point of view on this is that your repositorys commit history is a **record of what actually happened.** Its a historical document, valuable in its own right, and shouldnt be tampered with. From this angle, changing the commit history is almost blasphemous; youre *lying* about what actually transpired. So what if there was a messy series of merge commits? Thats how it happened, and the repository should preserve that for posterity.
- The opposing point of view is that the commit history is the **story of how your project was made.** You wouldnt publish the first draft of a book, and the manual for how to maintain your software deserves careful editing. This is the camp that uses tools like rebase and filter-branch to tell the story in the way thats best for future readers.

Summary

- Weve covered basic branching and merging in Git.
- You should feel comfortable creating and switching to new branches, switching between branches and merging local branches together.

- You should also be able to share your branches by pushing them to a shared server, working with others on shared branches and rebasing your branches before they are shared.