
--distributed-even-if-your-workflow-isnt

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

This book is translated into German, Chinese, French, Japanese, Dutch, Russian, Korean, Brazilian Portugese and Czech.

Partial translations available in Arabic, Spanish, Indonesian, Italian, Macedonian, Polish, Turkish, Taiwanese Mandarin, Spanish (Nicaragua), Catalan, Thai and Finnish.

Chapters ▾

3.4 Git Branching - Branching Workflows

Branching Workflows

Now that you have the basics of branching and merging down, what can or should you do with them? In this section, we'll cover some common workflows that this lightweight branching makes possible, so you can decide if you would like to incorporate it into your own development cycle.

Long-Running Branches

Because Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do. This means you can have several branches that are always open and that you use for different stages of your development cycle; you can merge regularly from some of them into others.

Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their `master` branch — possibly only code that has been or will be released. They have another parallel branch named develop or next that they work from or use to test stability — it isn't necessarily always stable, but whenever it gets to a stable state, it can be merged into `master`. It's used to pull in topic branches (short-lived branches, like your earlier `iss53` branch) when they're ready, to make sure they pass all the tests and don't introduce bugs.

In reality, we're talking about pointers moving up the line of commits you're making. The stable branches are farther down the line in your commit history, and the bleeding-edge branches are farther up the history (see Figure 3-18).

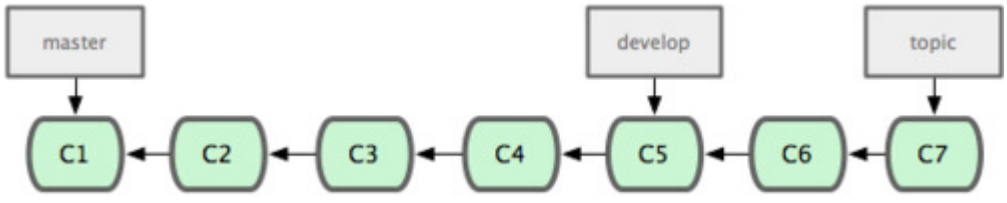

Figure 3-18. More stable branches are generally farther down the commit history.

It's generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they're fully tested (see Figure 3-19).

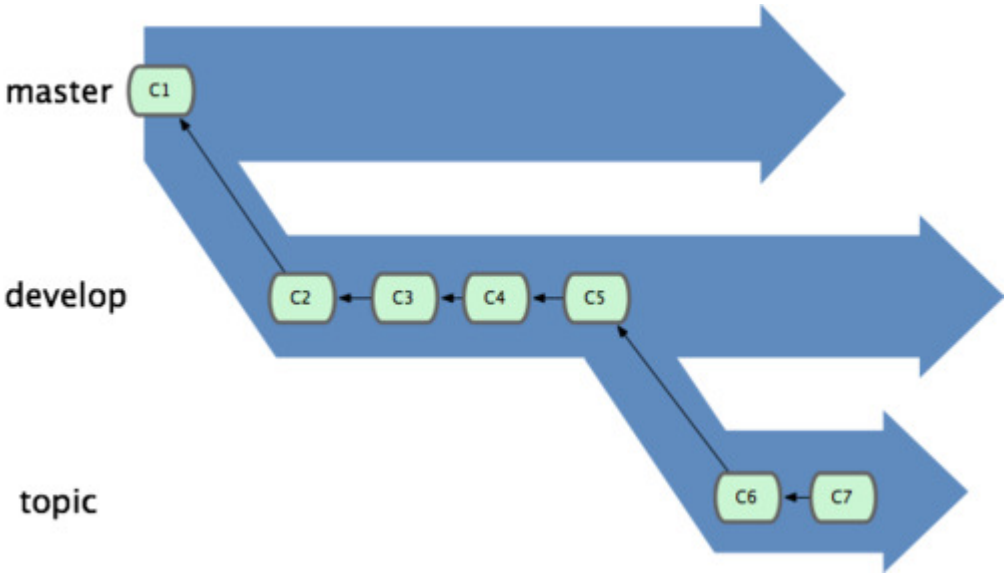

Figure 3-19. It may be helpful to think of your branches as silos.

You can keep doing this for several levels of stability. Some larger projects also have a `proposed` or `pu` (proposed updates) branch that has integrated branches that may not be ready to go into the `next` or `master` branch. The idea is that your branches are at various levels of stability; when they reach a more stable level, they're merged into the branch above them. Again, having multiple long-running branches isn't necessary, but it's often helpful, especially when you're dealing with very large or complex projects.

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. This is something you've likely never done with a VCS before because it's generally too expensive to create and merge branches. But in Git it's common to create, work on, merge, and delete branches several times a day.

You saw this in the last section with the `iss53` and `hotfix` branches you created. You did a few commits on them and deleted them directly after merging them into your main branch. This technique allows you to context-switch quickly and completely — because your work is separated into silos where all the changes in that branch have to do with that topic, it's easier to see what has happened during code review and such. You can keep the changes there for minutes, days, or months, and merge them in when they're ready, regardless of the order in which they were created or worked on.

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 you're not sure is a good idea (`dumbidea` branch). Your commit history will look something like Figure 3-20.

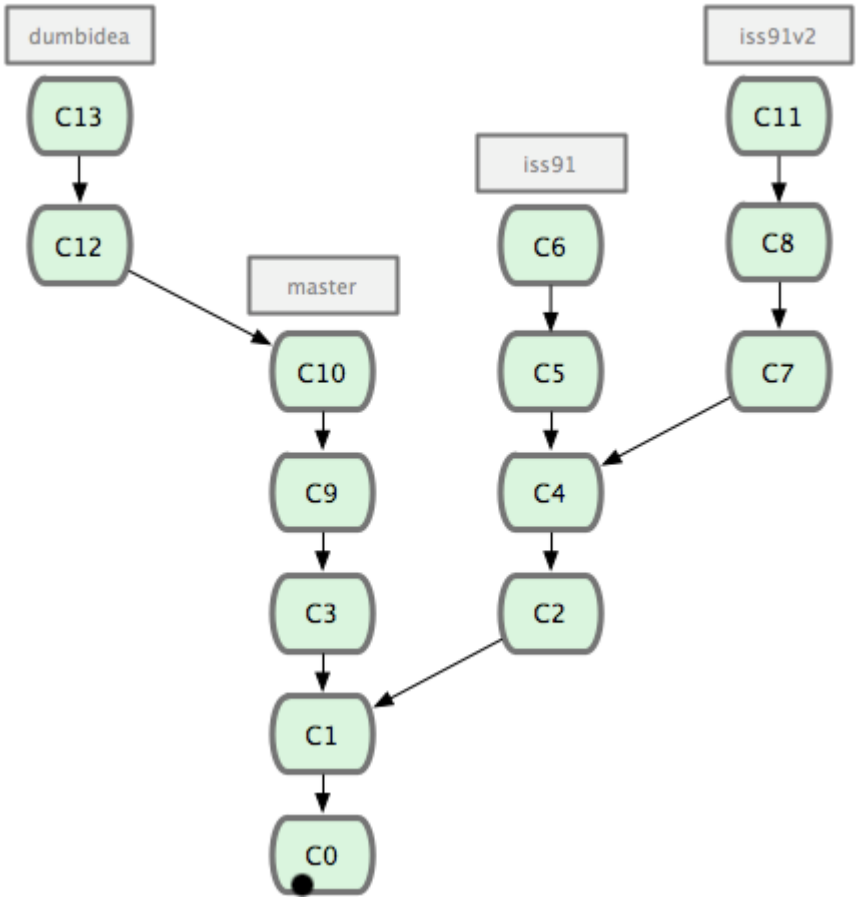

Figure 3-20. Your commit history with multiple topic branches.

Now, let's 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 Figure 3-21.

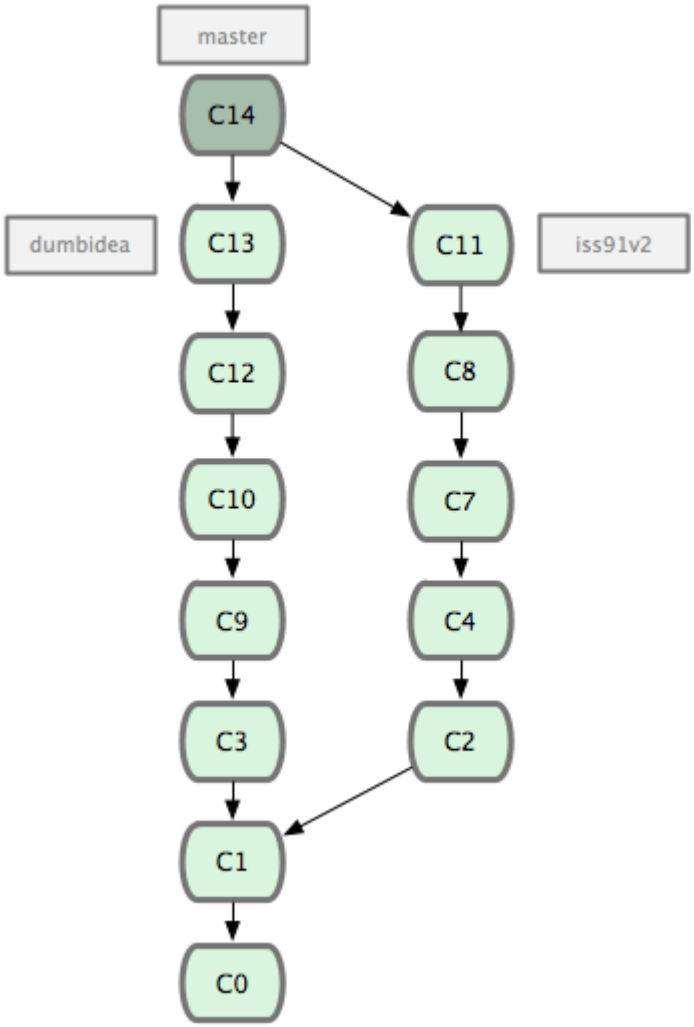

Figure 3-21. Your history after merging in dumbidea and iss91v2.

It’s important to remember when you’re doing all this that these branches are completely local. When you’re branching and merging, everything is being done only in your Git repository — no server communication is happening.

prev | next