Understanding Git: Branching

In This Section

The Purpose of Branching

Say you are working on a paper. You've gotten a first draft out, submitted for review. You then get a new batch of data, and you're in the process of integrating it into the paper. Halfway in, however, the review committee calls you up and tells you that you need to change some of your section headings to conform to format specifications. What do you do?

Obviously you don't want to send them your half-baked revisions with corrected headings. What you want to do is jump back to the version you sent out, change the headings on that version, and send off that copy, all the while keeping your recent work safely stored somewhere else.

This is the idea behind **branching,** and Git makes it easy to do.

Note on terminology: The terms "branch" and "head" are nearly synonymous in Git. Every branch is represented by one head, and every head represents one branch. Sometimes, "branch" will be used to refer to a head and the entire history of ancestor commits preceding that head, whereas "head" will be used to refer exclusively to a single commit object, the most recent commit in the branch.

Creating a Branch

To create a branch, say your repository looks like this:

```
(A) -- (B) -- (C)
               |
             master
               |
              HEAD
```

where (B) was the version you sent to the conference and (C) is your new revision state. (I'm dropping the arrowheads because they always point to the left.)

To jump back to commit (B) and start new work from there, you first need to know how to reference the commit. You could either use `git log` to get the SHA1 name of (B), or you could use *HEAD^* to retrieve it (as you will remember from the previous page, *HEAD^* means the parent of the *HEAD* commit).

Now, we use the `git branch` command:

```
git branch [new-head-name] [reference-to-(B)]
```

or, for example:

```
git branch fix-headers HEAD^
```

This command will create a new head with the given name, and point that head at the requested commit object. If the commit object is left out, it will point to *HEAD*.

Now our commit tree looks like this:

```
(A) -- (B) ------- (C)
        |           |
   fix-headers    master
                    |
                   HEAD
```

Switching Between Branches

In order to start working on the headers, you need to set the fix-headers head to be the current head. This is done with `git checkout`:

```
git checkout [head-name]
```

This command does the following:

- Points *HEAD* to the commit object specified by *[head-name]*
- Rewrites all the files in the directory to match the files stored in the new *HEAD* commit.

Important note: if there are any uncommitted changes when you run `git checkout`, Git will behave very strangely. The strangeness is predictable and sometimes useful, but it is best to avoid it. All you need to do, of course, is commit all the new changes before checking out the new head.

After checking out the *fix-headers* head, you fix the headers. Now you add and commit the changes as above. The resulting repository looks like this:

```
            +--------------- (D)
           /                  |
   (A) -- (B) -- (C)          |
                  |           |
                master  fix-headers
                              |
                            HEAD
```

(You can see now why it's called "branching": the commit tree has grown a new branch. Note that the angle of the line connecting (B) and (D) is irrelevant; pointers do not store whether they are horizontal or slanted.)

The ancestry of *master* is (C), (B), (A). The ancestry of *fix-headers* is (D), (B), (A). You can see this with `git log`.

Related Commands

Other useful commands at this point:

- `git branch` with no arguments lists the existing heads, with a star next to the current head.
- `git diff [head1]..[head2]` shows the diff between the commits referenced by *head2* and *head1*.
- `git diff [head1]...[head2]` (three dots) shows the diff between *head2* and the common ancestor of *head1* and *head2*. For example, `diff master...fix-headers` above would show the diff between (D) and (B).
- `git log [head1]..[head2]` shows the change log between *head2* and the common ancestor of *head1* and *head2*. With three dots, it also shows the changes between *head1* and the common ancestor; this is not so useful. (Switching *head1* and *head2*, on the other hand, is very useful.)

Common Branching Use Patterns

A common way to use Git branching is to maintain one "main" or "trunk" branch and create new branches to implement new features. Often the default Git branch, *master*, is used as the main branch.

So, in the example above, it may have been better to leave *master* at (B), where the paper was submitted to the reviewers. You could then start a new branch to store changes regarding new data.

Ideally, in this pattern, **the *master* branch is always in a releaseable state.** Other branches will contain half-finished work, new features, and so on.

This pattern is particularly important when there are multiple developers working on a single project. If all developers are adding commits in sequence to a single branch, then new features need to be added in a single commit, in order not to cause the branch to become unusable. However, if each developer creates a new branch to make a new feature, then commits can be made at any time, whether or not they are unfinished.

This is what Git users mean when they say that **commits are cheap.** If you are working on your own branch, there is no reason you need to be particularly careful about what you commit to the repository. It won't affect anything else.

Go on to the next section: Merging