Understanding Git: Merging

In This Section

- Merging
- Resolving Conflicts
- Fast Forward Merges
- Common Merge Use Patterns
- Deleting a Branch

Merging

After you have finished implementing a new feature on a branch, you want to bring that new feature into the main branch, so that everyone can use it. You can do so with the `git merge` or `git pull` command.

The syntax for the commands is as follows:

```
git merge [head]
git pull . [head]
```

They are identical in result. (Though the `merge` form seems simpler for now, the reason for the `pull` form will become apparent when discussing multiple developers.)

These commands perform the following operations. Let the current head be called *current*, and the head to be merged called *merge*.

1. Identify the common ancestor of *current* and *merge*. Call it *ancestor-commit*.
2. Deal with the easy cases. If the *ancestor-commit* equals *merge*, then do nothing. If *ancestor-commit* equals *current*, then do a **fast forward merge**.
3. Otherwise, determine the changes between the *ancestor-commit* and *merge*.
4. Attempt to merge those changes into the files in *current*.
5. If there were no conflicts, create a new commit, with two parents, *current* and *merge*. Set *current* (and *HEAD*) to point to this new commit, and update the working files for the project accordingly.
6. If there was a conflict, insert appropriate conflict markers and inform the user. No commit is created.

Important note: Git can get very confused if there are uncommitted changes in the files when you ask it to perform a merge. So make sure to commit whatever changes you have made so far before you merge.

So, to complete the above example, say you check out the *master* head again and finish writing up the new data for your paper. Now you want to bring in those changes you made to the headers.

The repository looks like this:

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

where (E) is the commit reflecting the completed version with the new data.

You would run:

```
git merge fix-headers
```

If there are no conflicts, the resulting respository looks like this:

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

The merge commit is (F), having parents (D) and (E). Because (B) is the common ancestor between (D) and (E), the files in (F) should contain the changes between (B) and (D), namely the heading fixes, incorporated into the files from (E).

Note on terminology: When I say “merge head A *into* head B,” I mean that head B is the current head, and you are drawing changes from head A into it. Head B gets updated; nothing is done to head A. (If you replace the word “merge” with the word “pull,” it may make more sense.)

Resolving Conflicts

A conflict arises if the commit to be merged in has a change in one place, and the current commit has a change in the same place. Git has no way of telling which change should take precedence.

To resolve the commit, edit the files to fix the conflicting changes. Then run `git add` to add the resolved files, and run `git commit` to commit the repaired merge. Git remembers that you were in the middle of a merge, so it sets the parents of the commit correctly.

Fast Forward Merges

A fast forward merge is a simple optimization for merging. Say your repository looks like this:

```
                   +-- (D) -- (E)
                  /            |
   (A) -- (B) -- (C)           |
                  |            |
               current     to-merge
                  |
                 HEAD
```

and you run `git merge to-merge`. In this case, all Git needs to do is set *current* to point to (E). Since (C) is the common ancestor, there are no changes to actually “merge.”

Hence, the resulting merged repository looks like:

```
                   +-- (D) -- (E)
                  /            |
   (A) -- (B) -- (C)           |
                               |
                       to-merge, current
                                    |
                                   HEAD
```

That is, *to-merge* and *current* both point to commit (E), and *HEAD* still points to *current*.

Note an important difference: no new commit object is created for the merge. Git only shifts the head pointers around.

Common Merge Use Patterns

There are two common reasons to merge two branches. The first, as explained above, is to draw the changes from a new feature branch into the main branch.

The second use pattern is to draw the main branch into a feature branch you are developing. This keeps the feature branch up to date with the latest bug fixes and new features added to the main branch. Doing this regularly reduces the risk of creating a conflict when you merge your feature into the main branch.

One disadvantage of doing the above is that your feature branch will end up with a lot of merge commits. An alternative that solves this problem is rebasing, although that comes with problems of its own.

Deleting a Branch

After you have merged a development branch into the main branch, you probably don't need the development branch anymore. Hence, you may want to delete it so it doesn't clutter your `git branch` listing.

To delete a branch, use `git branch -d [head]`. This simply removes the specified head from the repository's list of heads.

For example, in this repository from above:

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

we probably don't need the *fix-headers* head any more. So we can use:

```
git branch -d fix-headers
```

and the resulting repository looks like:

```
         +---------- (D) ---------------+
        /                                \
(A) -- (B) -- (C) -------------- (E) -- (F)
                                         |
                                       master
                                         |
                                        HEAD
```

Important note: `git branch -d` will cause an error if the branch to be deleted is not reachable from another head. Why? Consider the following repository:

```
         +----------- (E)
        /              |
(A) -- (B) -- (C)      |
               |       |
             head1   head2
```

Say you delete *head2*. Now how can you use commit (E)? You can't check it out, because it isn't a head.

And it doesn't appear in any logs or anywhere else, because it isn't an ancestor of *head1*. So commit (E) is practically useless. In Git terminology, it is a "dangling commit," and its information is lost.

Git does allow you to use the `-D` option to force deletion of a branch that would create a dangling commit. However, it should be a rare situation that you want to do that. **Think very carefully before using `git branch -D`.**

Go on to the next page: Collaborating