3.3 Git Branching - Branch Management

Branch Management

Now that you've created, merged, and deleted some branches, let's look at some branch-management tools that will come in handy when you begin using branches all the time.

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). This means that if you commit at this point, the `master` branch will be moved forward with your new work. 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 readme
```

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

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

Because you already merged in `iss53` earlier, you see it in your list. Branches on this list without the `*` in front of them are generally fine to delete with `git branch -d`; you've already incorporated their work into another branch, so you're not going to lose anything.

To see all the branches that contain work you haven't 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 isn't 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.

Tip The options described above, `--merged` and `--no-merged` will, if not given a commit or branch name as an argument, show you what is, respectively, merged or not merged into your *current* branch.

You can always provide an additional argument to ask about the merge state with respect to some other branch without checking that other branch out first, as in, what is not merged into the `master` branch?

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

```
topicA
featureB
```

Changing a branch name

Caution Do not rename branches that are still in use by other collaborators. Do not rename a branch like master/main/mainline without having read the section "Changing the master branch name".

Suppose you have a branch that is called *bad-branch-name* and you want to change it to *corrected-branch-name*, while keeping all history. You also want to change the branch name on the remote (GitHub, GitLab, other server). How do you do this?

Rename the branch locally with the `git branch --move` command:

```
$ git branch --move bad-branch-name corrected-branch-name
```

This replaces your bad-branch-name with corrected-branch-name, but this change is only local for now. To let others see the corrected branch on the remote, push it:

```
$ git push --set-upstream origin corrected-branch-name
```

Now we'll take a brief look at where we are now:

```
$ git branch --all
* corrected-branch-name
  main
  remotes/origin/bad-branch-name
  remotes/origin/corrected-branch-name
  remotes/origin/main
```

Notice that you're on the branch corrected-branch-name. The corrected branch is available on the remote. However the bad branch is also still present on the remote. You can delete the bad branch from the remote:

```
$ git push origin --delete bad-branch-name
```

Now the bad branch name is fully replaced with the corrected branch name.

Changing the master branch name

Warning Changing the name of a branch like master/main/mainline/default will break the integrations, services, helper utilities and build/release scripts that your repository uses. Before you do this, make sure you consult with your collaborators. Also make sure you do a thorough search through your repo and update any references to the old branch name in your code or scripts.

Rename your local *master* branch into *main* with the following command

```
$ git branch --move master main
```

There's no *master* branch locally anymore, because it's renamed to the *main* branch.

To let others see the new *main* branch, you need to push it to the remote. This makes the renamed branch available on the remote.

```
$ git push --set-upstream origin main
```

Now we end up with the following state:

```
git branch --all
* main
  remotes/origin/HEAD -> origin/master
```

```
  remotes/origin/main
  remotes/origin/master
```

Your local *master* branch is gone, as it's replaced with the *main* branch. The *main* branch is also available on the remote. But the remote still has a *master* branch. Other collaborators will continue to use the *master*branch as the base of their work, until you make some further changes.

Now you have a few more tasks in front of you to complete the transition:

- Any projects that depend on this one will need to update their code and/or configuration.
- Update any test-runner configuration files.
- Adjust build and release scripts.
- Redirect settings on your repo host for things like the repo's default branch, merge rules, and other things that match branch names.
- Update references to the old branch in documentation.
- Close or merge any pull requests that target the old branch.

After you've done all these tasks, and are certain the main branch performs just as the *master* branch, you can delete the *master* branch:

```
$ git push origin --delete master
```

prev | next