Using the Fork-and-Branch Git Workflow

27 January 2015 · Filed in Education

Now that I've provided you with an introduction to Git and a brief overview of using Git with GitHub, it's time to build on that knowledge by taking a closer look at one workflow often used when collaborating with Git. The "fork and branch" workflow is a common way of collaborating on open source projects using Git and GitHub. In this post, I'm going to walk through this workflow (as I understand it—I'm constantly learning), with a focus toward helping those that are new to this sort of thing.

If you're new to Git and/or GitHub and haven't yet read the earlier posts on Git and using Git with GitHub, I *strongly* recommend you read those first.

Basically, the "fork and branch" workflow looks something like this:

- Fork a GitHub repository.
- Clone the forked repository to your local system.
- Add a Git remote for the original repository.
- Create a feature branch in which to place your changes.
- Make your changes to the new branch.
- Commit the changes to the branch.
- Push the branch to GitHub.
- Open a pull request from the new branch to the original repo.
- Clean up after your pull request is merged.

Here's a bit more detail on each of these steps in the workflow.

Forking a GitHub Repository

The first step is to fork the GitHub repository with which you'd like to work. For example, if you were interested in helping contribute content to the Open vSwitch web site, which is itself hosted as a GitHub repository, you would first fork it. Forking it is basically making a copy of the repository, but with a link back to the original.

Forking a repository is *really* straightforward:

1. Make sure you're logged into GitHub with your account.
2. Find the GitHub repository with which you'd like to work.
3. Click the Fork button on the upper right-hand side of the repository's page.

That's it—you now have a copy of the original repository in your GitHub account.

Making a Local Clone

Even though you have a copy of the original repository in your GitHub account, you don't yet have a way to make changes to your copy of the repository. Recall from the two earlier Git/GitHub articles that you first have to clone the repository down to your local system using `git clone`. Otherwise, you have no way of actually making changes to the forked repository in your GitHub account.

Before you can run `git clone`, though, you need to determine the URL for the forked repository. That's pretty simple: just navigate to the forked repository (this is the copy of the original repository residing in your GitHub account) and look on the right-hand side of the web page. You should see an area that is labeled "HTTPS clone URL". Simple copy the URL there, and then use it with `git clone` like this (this is the clone URL for the OVS web site repository):

```
git clone https://github.com/openvswitch/openvswi
tch.github.io.git
```

Naturally, you'd replace the URL after `git clone` with the appropriate HTTPS clone URL for your forked project

repository. (You have to clone your forked repository, **not** the original.)

I've talked about what happens with `git clone` before, so I will only very quickly review what happens when you clone the forked repository. Git will copy down the repository, both contents and commit history, to your system. Git will also add a Git remote called `origin` that points back to the forked repository in your GitHub account.

If you were only interested in making a fork of the project and not contributing back to the original project, you could stop here. You could continue development on this fork of the original project using the workflows and commands I described in the previous two articles. However, the real goal of this article is to help you contribute back to a project, so you need to keep going.

Adding a Remote

Git already added a Git remote named `origin` to the clone of the Git repository on your system, and this will allow you to push changes back up to the forked repository in your GitHub account using `git commit` (to add commits locally) and `git push`. I described this process in the previous article on using Git with GitHub.

Recall that the generic form of `git push` is this `git push <remote> <branch>`; this implies that there could be other Git remotes besides `origin`. That is definitely the case; as I mentioned in the previous article, you'll need a Git remote for any repository to which you'd like push changes or from which you'd like to pull changes.

To use the "fork and branch" workflow, you'll need to add a Git remote pointing back to the **original** repository (the one you forked on GitHub). You can call this Git remote anything you want; most documents and guides I've read recommend using the term "upstream". Let's say you forked the OVS web site repo, and then cloned it down to your system. You'd want to add a Git remote that points back to the original repository, like this:

```
git remote add upstream https://github.com/openvs
witch/openvswitch.github.io.git
```

Of course, you'd want to replace the URL after `git remote add upstream` with the appropriate clone URL for the original project repository. For the rest of this article, I'm going to assume you used `upstream` as the name for this additional Git remote.

Now, you *could* try pushing changes to the original repository using `git push` at this point, but it would probably fail because you probably don't have permission to push changes directly to the repository. Besides, it **really** wouldn't be a good idea. That's because other people might be working on the project as well, and how in the world would we keep track of everyone's changes? That's where branches come in.

Working in a Branch

So far, you've forked a project repository, cloned it down to your local system, and added a Git remote to your clone that points to the original project repository on GitHub. Now you're ready to start making changes to your local Git repository. You can't just start making changes willy nilly, though; to effectively collaborate with others on the same repo, you should use a *branch*.

There's a lot more to Git branches than I have room to discuss here (and a lot more that I have yet to learn), but suffice it to say for now that this is a way to create a separate line of changes (a "branch") that is independent from the main line (often referred to as "master"). There are lots of reasons to use branches in your development, but for the purposes of this discussion on using the "fork and branch" workflow to collaborate with others, the purpose of a branch is to help facilitate multiple users making changes to a repository at the same time.

So, assuming that your goal is to issue a pull request to change your changes merged back into the original project, you'll need to use a branch. Often you'll see this referred to as a *feature branch*, because you'll typically be implementing a new feature in the project.

The basic flow looks something like this (all this is happening on your local Git repository):

1. Create and checkout a *feature branch*.
2. Make changes to the files.
3. Commit your changes to the branch.

Because of the way that Git works, it's incredibly fast and easy for developers to create multiple branches. If you're relatively new to Git (as I assume you are if you're reading this), then you'll probably want to take it easy and not go totally crazy with branches. As I am ramping up my familiarity with Git, I'm sticking to a single feature branch at a time.

To create a new branch and check it out (meaning tell Git you will be making changes to the branch), use this command:

```
git checkout –b <new branch name>
```

Note that some projects have specific requirements around branch names for pull requests, so be aware of any such guidelines. Once the new branch is created and checked out, then you can make the necessary changes in this branch to implement the specific feature or change you want the original project to merge into their repository. As a general rule of thumb, you should limit a branch to *one logical change.* The definition of one logical change will vary from project to project and developer to developer, but the basic idea is that you should only make the necessary changes to implement one specific feature or enhancement.

As you make changes to the files in the branch, you'll want to commit those changes, building your changeset with `git add` and committing the changes using `git commit` . Also be aware that some projects don't want a bunch of commits in a pull request, so you may need to use `git rebase` to "squash the commit history". To keep this article manageable, I won't try to cover that here, but there are numerous guides on how this works (here is one such guide on using `git rebase` to squash commits).

Pushing Changes to GitHub

So let's say you've made the changes necessary to implement the specific feature or enhancement (the one "logical change"), and you've committed the changes to your local repository. The next step is to push those changes back up to GitHub.

If you were working in a branch called `new-feature`, then pushing the changes you made in that branch back to GitHub would look like this:

```
git push origin new-feature
```

Again, recall that the generic form of this command is `git push <remote> <branch>`. In this case, you're pushing changes in the `new-feature` branch to the `origin` remote.

Opening a Pull Request

GitHub makes this part incredibly easy. Once you push a new branch up to your repository, GitHub will prompt you to create a pull request (I'm assuming you're using your browser and not the GitHub native apps). The maintainers of the original project can use this pull request to pull your changes across to their repository and, if they approve of the changes, merge them into the main repository.

Be aware that, to help keep things manageable, some open source projects may have guidelines around how pull requests are submitted. I mentioned earlier that you might need to use a certain name for your feature branch, or perhaps the project requests that you create an issue (another GitHub feature) before submitting the pull request (and then including the issue number in the pull request). Just check with the maintainers of the particular project(s) to which you'd like to contribute to see if any such details exist.

Cleaning up After a Merged Pull Request

If the maintainers accept your changes and merge them into the main repository, then there is a little bit of clean-

up for you to do. First, you should update your local clone by using `git pull upstream master`. This pulls the changes from the original repository's (indicated by `upstream`) master branch (indicated by `master` in that command) to your local cloned repository. One of the commits in the commit history will be the commit that merged your feature branch, so after you `git pull` your local repository's master branch will have your feature branch's changes committed. This means you can delete the feature branch (because the changes are already in the master branch):

```
git branch -d <branch name>
```

Then you can update the master branch in your forked repository:

```
git push origin master
```

And push the deletion of the feature branch to your GitHub repository (*update:* an earlier version of this article listed `git push -d` below):

```
git push --delete origin <branch name>
```

And that's it! You've just successfully created a feature branch, made some changes, committed those changes to your repository, pushed them to GitHub, opened a pull request, had your changes merged by the maintainers, and then cleaned up. Pretty neat, eh?

Keeping Your Fork in Sync

By the way, your forked repository doesn't automatically stay in sync with the original repository; you need to take care of this yourself. After all, in a healthy open source project, multiple contributors are forking the repository, cloning it, creating feature branches, committing changes, and submitting pull requests.

To keep your fork in sync with the original repository, use these commands:

```
git pull upstream master
git push origin master
```

This pulls the changes from the original repository (the one pointed to by the `upstream` Git remote) and pushes them to your forked repository (the one pointed to by the `origin` remote). (Hopefully that makes a little bit of sense to you by now.)

Additional Resources

I found the Help on the GitHub site to be particularly useful.

Tags: CLI · Collaboration · Git · Linux

Previous Post: Using Git with GitHub

Next Post: Why Comments Aren't Available Yet

Be social and share this post!

Recent Posts