More Git Functionality

Jared Bennett

19 November, 2019

Useful References

Berkeley SCF Git Basics
Basic Branching and Merging
Interactive Branching Tutorial
Advanced Merging Undoing Things

Learning Objectives

- Finding help
- Importing a new project into Git
- Making new branches
- Making changes/merging to a git repo

Manual Pages

You can get documentation for a command such as `git log --graph` with:

```
man git-log
```

or

```
git help log
```

Importing A Project

I **do not** recommend this process for initiating a new project. These steps are simple, until you get to creating the remote repository. Then, just like in the intro tutorial, you have you setup a new repository on Github and link it to the local one.
It is easier to create the repo on Github, clone the empty repo locally, then put files in it as desired.

Assume you have a tarball linReg.tar.gz with your initial work. You can place it under Git revision control as follows.

```
tar xzf project.tar.gz
cd project
git init
```

Git will reply (something along the lines)

```
Initialized empty Git repository in .git/
```

You've now initialized the working directory-you may notice a new directory created, named ".git".

Next, tell Git to take a snapshot of the contents of all files under the current directory (note the .), with git add:

```
git add .
```

This snapshot is now stored in a temporary staging area which Git calls the *index*. You can permanently store the contents of the index in the repository with `git commit`:

```
git commit -m "add"
```

This will prompt you for a commit message. You've now stored the first version of your project in Git.

Making Changes

```
git add file1 file2 file3
```

You are now ready to commit. You can see what is about to be committed using `git diff` with the `--cached` option:

```
git diff --cached
```

(Without –cached, git diff will show you any changes that you've made but not yet added to the index.)

You can also get a brief summary of the situation with `git status`:

```
git status
```

Alternatively, instead of running `git add` before `git commit`, you can use:

```
git commit -a
```

which will automatically notice any modified (but not new) files, add them to the index, and commit, all in one step.

Git Tracks Contents, Not Files

Many revision control systems provide an `add` command that tells the system to start tracking changes to a new file. Git's `add` command does something simpler and more powerful: `git add` is used both for new and newly modified files, and in both cases it takes a snapshot of the given files and stages that content in the index, ready for inclusion in the next commit.

Undoing a Mistake

What if, in the files you just commited, there was a file you forgot?
This situation is handled via git's `amend` option in git commit.

```
man git-commit
```

This allows you to add more files to a commit and then update the message, while keeping your original message/commmited-files there.

However, if you added something that shouldn't be commited, then you need to use the `git reset` feature.

```
man git-reset

# e.g.
git reset --soft HEAD~1
git reset --hard HEAD~1
```

Git reset moves the tip of your working tree back to the specified revision (here, we go back one revision). The `--soft` flag means that the changes in the files are presevered, so all that was done was to undo the commit. If you use the `--hard` flag, then all changes are reverted to the specified time.

If you have done something more complex, like commiting and attempting to push a large file, the command below may be useful.

```
git filter-branch --index-filter 'git rm -r --cached --ignore-unmatch <PATH/TO/FILE>' HEAD
```

This deletes everything in the commit history.

Viewing Project History

At any point you can view the history of your changes using:

```
git log
```

If you also want to see complete diffs at each step, use

```
git log -p
```

Often the overview of the change is useful to get a feel of each step:

```
git log --stat --summary
```

For a prettier, more detailed graph (with several more options to look up):

```
git log --oneline --decorate --graph --all
```

Managing Branches

A single Git repository can maintain multiple branches of development. To create a new branch named "experimental", use

```
git branch experimental
```

If you now run

```
git branch
```

The "experimental" branch is the one you just created, and the "master" branch is a default branch that was created for you automatically. The asterisk marks the branch you are currently on; type

```
git checkout experimental
```

to switch to the experimental branch. Now edit a file, commit the change, and switch back to the master branch:

```
git commit -a "edit"
git checkout master
```

Alternatively, one can "stash" the changes using `git stash`.
This saves your changes for later, and then reverts the working tree to the last HEAD (whatever the last commit was). This allows you to keep working without the changes being applied to any files.
You can apply those changes later using `git stash pop`, which applies the changes and removes them from your stash. Or, if you wish to apply the changes to multiple branches, you can use `git stash apply`, which applies the changes but leaves them in your stash.

Now, you can switch back to the master branch.

```
git checkout master
```

Check that the change you made is no longer visible, since it was made on the experimental branch and you're back on the master branch.

You can make a different change on the master branch and commit. At this point the two branches have diverged, with different changes made in each. To merge the changes made in experimental into master, run:

```
git merge experimental
```

If the changes don't conflict, you're done. If there are conflicts, markers will be left in the problematic files showing the conflict;

```
git diff
```

will show this. Once you've edited the files to resolve the conflicts,

```
git commit -a
```

will commit the result of the merge. Finally,

At this point you could delete the experimental branch with

```
git branch -d experimental
```

This command ensures that the changes in the experimental branch are already in the current branch.

If you want to remove a branch without pulling the changes into the master branch, the `-D` flag deletes it without checking any of the changes.

At some point before removing a branch, you may want to merge the branch back into the master branch. The best way to do this is using a pull request, which I will demonstrate on github. Pull requests tell the repository maintainers what changes you are proposing, whether it can be directly merged in or if there are conflicts, and cleanly wraps the log from the other branch into one statement. You can also set a default template for your pull requests.

Using Git For Collaboration

We will do this as a partner exercise, using several of the things we have already seen and the example document in provided in the repo.

1) Find a partner (i.e., the person next to you, groups of 2 only)
2) Create a new repo (just one of you, doesn't matter who)
3) Add your partner as a collaborator
4) I want everyone to test running-into and resolving 2 kinds of merge errors:
    - What happens when the remote repository is ahead of your local one, but you have already staged new changes?
    - What happens when a collaborator (or you on a different computer) edits the same file?