CS 246E Fall 2021 — Tutorial 5

October 4, 2021

Summary

1 What is version control

Often when writing code we want to be able to look back at old versions of code, whether because we made a mistake when changing something and want to roll back, or simply want to remember what code used to look like. Similarly, we want the ability to manage multiple versions of a codebase (owned by different people), and to synchronize them between computers. Version control helps with these tasks.

There have been a number of popular version control tools over the years, including (but not limited to)

- Subversion (svn)
- Git
- Mercurial
- Concurrent Versions System (cvs)

Currently git is arguably the most famous and popular, in no small part due to the support of `github.com`, but all have seen plenty of use. We will focus on git for this tutorial.

2 Using git: the basics

To use git you'll need a repository. While we could use the course repository, you don't have permissions to do some operations on it, so we'll make our own

1. First, navigate into whatever folder you want your repository to live in. Type `git init` to create an empty local git repository
2. At this point, create some files. Once you've done so, you can ask git to start tracking them (or notice an update of a file that's changed) using `git add`. If you `git add` a folder, it will recursively add every file in the folder
3. Once you are ready to store a revision (version) of your code long-term, use `git commit` to have git remember the current state of any files you added to it since your last commit. You should supply a short "commit message" so that someone looking at your commits later can tell what changed here. To do this, you can use `git commit -m "Your message here"`. If you don't use `-m`, git will open $EDITOR and ask you to enter a commit message
4. If you want to move a file to a different folder and have git recognize this, delete a file from the repository, or any other usual filesystem task, use `git mv`, `git rm`, etc. Note that these both perform the actual action you'd expect (moving the file, deleting the file, etc) while also telling git that it's happened so that the repository can be updated next time you commit

3 Working on multiple computers: setting up git

You can share a repository among multiple computers as long as one of them hosts the repository: it's easiest to do this by using a website such as github. For assignments it's best not to use github unless you have access to private repositories since otherwise you may inadvertently share your assignment solutions with other students, but luckily UW has their own service (UW gitlab) at `https://git.uwaterloo.ca/` for this purpose. You can have up to 10 private repositories at a time, which should be more than enough.

1. First, try creating a repository on gitlab
2. You can add a remote repository to your machine by using git clone followed by the repository's URL. Most services (such as github and gitlab) should give you a link to copy. Try cloning your repository on to your machine
3. You can send your commits to any other people using your repository by using `git push`. Try creating some files, committing them, and then pushing them. You should now be able to see them on gitlab
4. If you want to receive any changes other users have made, you can use `git fetch` or `git pull` to receive other peoples' changes. We'll talk about the difference later. For now, try cloning the repository into a second location and using git push/pull to share some files

4 Preventing committing the wrong files

There are lots of files you may not want to track in your git repository. For example, usually you don't want to track PDFs and intermediate files related to tex documents, binaries, object files, dependency files, temporary vim/emacs files and so forth. While we could just avoid adding them using `git add`, unfortunately these files will still be added if we add directories.

Luckily there's a more convenient way to avoid adding certain files you don't want. Create a file called `.gitignore` in the root directory of the project and commit it. It should contain a list of files to ignore, one per line. You can use globbing patterns, so for example you can tell git to ignore all object files by putting `*.o` in the `.gitignore` file.

Try creating a `.gitignore` file which excludes vim swap files (files beginning with a `.` and ending in `.swp`) and the file `a2q3b`, then test it out by putting some files in a directory, some of which are swap files or named `a2q3b`, and adding them to the repository.

5 Merge conflicts

A merge conflict occurs when the code in your repository disagrees with the repository you're pulling from. Once you have two or more people working on the same project you may start running into merge conflicts. Even if you're the only person working on your project, if you are working on multiple machines you may occasionally forget to commit or push changes, which may result in merge conflicts anyways.

1. Try changing a file on two copies of your repository. Commit and push one of them
2. Commit your file on the other copy of your repository and try using `git pull`. Git will merge the two versions and commit histories, which will likely require you to manually synchronize any file which was modified by both sides
3. Alternatively, you can use `git fetch` to receive remote changes without alternatively attempting to merge. If you do, you will have to manually merge with `git merge` afterwards. You must merge before

you can `git push` your commits to other copies of the repository

Merging can be annoying and occasionally difficult. Whenever possible you should try to commit early and often, pull before you make changes, and try not to work on the same file between multiple machines or people unless absolutely necessary. That being said, sometimes it's unavoidable, and accidents happen.

6 Retrieving old versions of files

Occasionally you may want to roll back parts of your code (or all of it) to a previous version, whether because you created a bug which you can't identify, because you deleted a file and now need it back, or for any number of other reasons.

1. Delete a file from your repository using `git rm`. Commit and then make another few changes and commit each time
2. Now we want to get the file back (the same process works to restore old versions of existing files). View the recent commit history using `git log`
3. Once you've found the commit you're looking for (this is one of the places where commit messages come in handy!), note its commit number. We can now check out the file we want from the version we want via `git checkout [commit number] [file]`. This doesn't change the state of the repository: you still need to re-add and commit it if you want the changes to be recognized long-term

7 Collaborating and rolling back more effectively: branches

As you can see, merging and retrieving old files both work, but they're a bit clunky. Constant merging when collaborating with someone (or alternatively coordinating to make sure you're never changing the same files) gets annoying and time-consuming and can be a major source of bugs. Identifying which commit holds the version of a file you want can be difficult, especially if you need to roll back a major feature, code refactor or other change. Git provides branches to make this process easier.

You may have noticed that everything git has said so far tends to have the word `master` on it somewhere or other. This is the default branch. A branch is essentially a version of the repository: anyone on the same repository has access to every branch, but at any one time you're only working on a single branch and your changes don't affect anyone working on a different branch than you are.

1. Create a new branch of your repository using `git branch [branch name] [old branch name]`, which creates a new branch starting from the old branch. You can move to the branch using
   `git checkout [branch name]`
2. Try making some changes to the new branch, committing them and pushing them
3. In the other copy of your repository, make some changes to master, commit them and push them. Notice how you have no merge conflicts
4. In the original copy of your repository where the branch was made, switch to the master branch using `git checkout master`, pull all the changes made to master, then use `git merge [branch name]` to merge your branch with master. This lets you do all of the merging at once and have well-defined places where features are incorporated into the main codebase. In serious software projects all development is strongly encouraged to happen on branches rather than on master: whether you choose to follow this guideline is up to you

8 ...and much more

Git has many more commands, features, options to commands and so forth than we have time to explore in this short tutorial. You can find plenty of tutorials online, and can view git's manual using `man git` and `man git-[command]` (for example, `man git-checkout`).

You should try using git or another VCS to manage your code in CS246E from now on as well as your personal projects, since version control is going to be a mainstay of your life from here on out!