Version Control using Git

January 21, 2019

Version control using Git

Sometimes you make mistakes and want to revert to an old version of the code, or you want to explore some features that may or may not end up belonging to the project, or you want to cooperate with other people and be able to selectively incorporate their changes to the codebase. Version control systems help with all these tasks. A version control system is essentially a way to keep code organized that lets you go back to different versions of the file and gives you tools to merge different versions from different authors.

The benefits of version control are many: * It allows you to keep files safe. If you accidentally delete something, you can get your old work back as long as you `committed` the old work at some point.

- No need to comment-out old code (for a rainy day, right?). You can simply delete the unneeded code, and if you really do need it again, it can be recovered.
- You can work with a clean working directory. No need to have five different versions of a source file floating around. With version control, you still have multiple versions, but they are hidden away from you, so they do not add clutter or confuse you.
- You can work on multiple features in parallel and easily merge them. Similarly, it allows for easier collaboration between multiple programmers.

git is very popular but not the only alternative. git is "distributed" which means that instead of having a centralized codebase with which each user interacts,each peer's working copy of the codebase is a complete repository.

git has a reputation for being complex, but it is becoming the standard in version control. Also, most of the tasks in a normal operation with git only involve a few, very intuitive commands.

We start using git by setting up a repository: we can either start a new project in our folder or we can pull code from an existing repository. Let's start by making our own new repo. In the current directory, we run

```
git init
```

which will create a .git folder. We can check the status of our repository with

```
git status
```

We can then add files to the repostory using git add filename. filename is now added to the staging area and the changes will be tracked.

The changes are committed to the repository with

```
git commit -m "Adding files"
```

The `-m` option allows us to include a message describing what we did in this commit. Running `git commit` will do nothing unless you explicitly add files to the commit with `git add`.

Commits effectively serve two purposes. Firstly, they represent a "save point" in your work. You can always restore your work to any point in history where you commited the file(s). Secondly, through the use of messages, commits tell a story, so you can see how your project developed over time and recover old files as needed based on that story. Because of the latter purpose, each commit should represent a complete thought, such as a bug fix or a new feature.

We can now take a look at the history of changes using

```
git log
```

The changes so far have been stored in our local copy of the repository, but we usually want to store a copy of our code in a remote server. To do that, we can push the changes to a particular address (either https or ssh). We can register this remote with a label using

```
git remote add origin https://example.com/myproject.git
```

Now, we can send the code to the master branch of our code in the origin remote server with

```
git push origin master
```

How do we retrieve upstream changes? Imagine some has made changes to the code from their side and we want to incorporate them to our copy of the repository. We can then

```
git pull origin master
```

to download the changes to the master branch in the origin server. This operation will fetch the changes and merge them into the codebase automatically unless there is a conflict.

Branches

One of the most important features of Git is branches. Instead of a linear set of changes, you may wish to develop multiple features in parallel and merge them later. Branches allow you to do that. At any point in your commit history, you can split the history into two or more branches. Once each branch has served its purpose, it can be merged with your main branch. Git handles merging automatically as well as it can by comparing the differences in the files of both branches, but there are occasions where you will have a "merge conflict" and have to resolve conflicts manually.

You can create and switch to a branch like this:

```
git checkout -b my_branch_name
```

If the branch already exists, then you may leave off the `-b` option.

To merge the changes of one branch to another, switch to the target branch (via `git checkout`) and use `git merge other_branch_name`.

Stashes

Recall that commits serve two purposes: to save your work and to tell a story via commit messages, each representing a thought. Sometimes you only want the former. One example is when you are working on a feature, and you receive an emergency request, but the feature you are working on is not complete. You want to save your work and switch to another branch to handle the emergency request, but you don't want a full commit because what you have done doesn't represent a complete thought, so would pollute the commit history. The `git stash` command is made for this. It saves your work just like a commit, but it is effectively temporary. Use `git stash` to save your work and use `git pop` to apply your saved work to the current branch (see also: `git apply`).

Final note

Version control software is used by primarily by programmers for control of source code. However, there is no reason its use should be limited to this. For example, if you are creating a report in Word, you can store the files and track the history of changes via Git. However, be careful to note that some types of files, such as

pdfs, can be difficult to merge, so use of version control to develop files in parallel is most useful with simple text files, e.g., source code.