Version Control and Teamwork

Michael Walker

31st January, 2013

Contents

1 Version Control

A version control system (VCS) is a tool (or collection of tools) which make working on multiple versions of something, be it a program, website, document, whatever, and keeping track of all of the changes much easier. If it helps, you can think of a VCS as a glorified undo tool.

If you're used to developing software by first making a backup of the file you want to work on, and then editing the original file (or even just editing the original file); or by editing the live system rather than an offline copy; you need version control.

The work-flow for version control is a bit different:

Edit the appropriate files, there is no need to take a backup first.

Test your changes. Of course, you did this before, right?

Commit your changes, that is, tell the VCS which files you have changed and (usually) enter a short message explaining what you did.

Push your commits, that is, send your commits to a server, or some other user. If you're the sole user of the VCS for your project, this may not be necessary (but can still be good—more on that later).

You may think that having to enter a message for each commit is a hassle. Well, consider that by entering a good message, you'll be able to see what the commit did. This may be incredibly useful down the line when you're trying to find out where you introduced a bug, or removed something that was needed. By using a VCS, you're not just giving yourself an easy way to undo things, you're building a detailed history of the development of your project, which is an invaluable tool in itself.

Eventually, you will make a commit that you realise was a bad idea. It's inevitable. It's ok. You can tell people you were drunk when you did it. Fortunately, we come to another strength of VCSs here, undoing things. Every VCS (that I know of) has some way to tell the system to undo a specific commit. How this is implemented varies—the commit may simply vanish, or the system may insert a new commit reverting all of its changes. You may also need to do some manual intervention, if you're undoing something which you have since built upon.

Finally, let's cover branches and merging in this introduction. These are key concepts in the VCS workflow, and entire development paradigms have sprung up around how to use branches correctly (we'll be looking at two). A branch is a copy of your code. Each branch can be distinct, and branches

can be merged (perhaps with some manual intervention required) at any time as well. It's possibly easiest to explain with a couple of examples of typical branches you may have:

master this is typically your stable, production-ready code.

devel depending on your workflow, you may also have a development branch. This will be ahead of master, and will be merged into master when you have implemented everything you want for the next release

hotfixes again, depending on your workflow, you may have a branch for emergency fixes to releases. This will also be ahead of master, but only by a handful of commits at a time, after which point it will be merged and a new release made.

Of course, where branches really come into their own is when you have a good workflow based around them. It's also typical to have "feature branches", one branch for each new feature you are implementing, which is them merged into the development branch when complete.

1.1 Distributed vs. Centralised

There are two approaches when it comes to designing version control systems, and which one your system uses will drastically affect how you work with it. These approaches are distributed (or decentralised) and centralised systems.

Centralised systems have one central server which houses the repository. If you commit something, it is sent to the server, if you lose access for a period of time you can't keep working and push your commits later. Despite this downside, because everyone is working off the same copy of the code, conflicts when merging may be reduced.

Distributed systems do not have a central server. If you commit something, it is saved on your local machine. When you push your commits, that simply consists of you sending them to another user, who merges them into their copy. As there is no server to lose access to, you can keep working at all times. However, because everyone has their own version of the code, which may have many commits difference with someone else, merge conflicts may be more common.

In practice, distributed systems are much more convenient to use than centralised systems, and the potential problem of very inconsistent repositories is not often realised. Furthermore, users typically use a hybrid of the

two approaches, a distributed system with one "canonical" version of the code, which may be guarded by one privileged user. There may even be multiple levels of this canonical code, forming a tree structure with trusted users at each level accepting things from the level below which pass the quality checks, possibly doing some more work of their own, and then pushing their code up the tree to be reviewed by the next person. This sort of structure is often used in large open source projects, for example, the Linux kernel. Linus Torvalds has control over the canonical kernel code, and reviews all commits to it and either accepts or denies.

A typical workflow for distributed systems is to have an account on a service like GitHub, which provides a globally-accessible repository which you can push your code to. This allows you to work anywhere even if you didn't bring your local copy of the code with you, provides free backup of all of your commits, and enables people working with you to frequently merge your commits and so reduce potential conflicts.

1.2 Some popular VCS tools

There are a lot of version control tools. Below are some of the major ones, which is it often convenient to have at least a basic knowledge of, even if you use one of them exclusively for everything you do:

Concurrent Versions System (CVS) centralised, very old.

Subversion (svn) centralised, intended to be a better CVS.

Git distributed, has gained a lot of popularity in the open source community.

Mercurial (hg) distributed, has gained some popularity in the open source community.

I find that centralised tools are usually used in organisations with a strict hierarchy, and distributed tools in organisations where every developer is (more or less) equal. If nothing else, an interesting modern manifestation of how the Cathedral and the Bazaar methods differ.

If you're going to join us in working on the DDNS project, we shall be using Git and GitHub extensively.

2 Using Git

This is just going to be a quick command reference, if you want more details, you can read the manpages or look online.

`git init` Initialise a new git repository in the current directory.

`git add` Tell git about a changed file or directory.

`git mv` Move a file

`git rm` Delete a file

`git commit` Commit a set of changes

`git log` Look at the git history

`git checkout` Create and change branches

`git remote` Manage remote repositories (*e.g.* your github repo)

`git push` Push commits to a remote

`git fetch` Fetch (but do not merge) changes from a remote

`git merge` Merge another branch (including things fetched from a remote).

`git tag` Give a name (*e.g.* a version number) to the latest commit.

Now, we're going to look at two workflows common with git, which you use depends on both personal preference and how the project you are working on is structured.

2.1 Git Flow

Git Flow is a method of development with Git for projects that have a clear release schedule, where one branch is to be used to contain the production-ready code. It is a good methodology to use when working with a lot of other people, or on some large or important system, as it is very strict about how development and releases are done.

There are two main branches that always are present:

master the main branch, containing production-ready code. Every commit should be a new release.

develop the main development branch, containing code for the next release.

`develop` is, of course, almost always ahead of `master`. When the code in `develop` reaches a point suitable for a new release, it is merged into `master`.

Git Flow uses a number of support branches: feature branches, release branches, and hotfix branches. Each one can only be branched off from certain branches and can only merge back into certain other branches. This rigid structure helps to ensure development progresses smoothly.

2.1.1 Feature branches

These can only branch off from `develop` and can only be merged back into `develop`.

These are used to develop features for some future release, although which release may be unknown. The idea is that once a feature is done, it will be merged into `develop` when it is desired to go into the next release (it will wait until that time) or, if the feature doesn't work out, the branch will just be deleted.

A feature branch is created by branching from `develop`:

```
$ git checkout -b FEATURE develop
```

And, when finished, will be merged back into `develop`:

```
$ git checkout develop
$ git merge --no-ff FEATURE
$ git branch -d FEATURE
$ git push origin develop
```

The flag `--no-ff` causes the merge to be stored as a new commit, even when this is unnecessary. The reason for this is that it is easier to see in the history when a feature branch starts and when it stops, which makes removing a feature (if necessary) much easier.

2.1.2 Release branches

These can only branch off from `develop` and must merge back into both `develop` and `master`.

These are used to prepare a new release. When the code in `develop` is in a suitable state, a release branch is created and the first commit to the branch is updating the version number. After that, there may be some number of commits to finalise the code for release. Once it has been finalised, it is merged into `master` (creating a new release) and `develop`.

A release branch is created by branching from `develop`:

```
$ git checkout -b release-VERSION develop
update the version number
$ git commit -a -m "Updated the version number to VERSION"
```

And, when finished, will be merged into both `master` and `develop`:

```
$ git checkout master
$ git merge --no-ff release-VERSION
$ git tag -a VERSION
$ git checkout develop
$ git merge --no-ff release-VERSION
$ git branch -d release-VERSION
```

2.1.3 Hotfix branches

These can only branch off from `master` and must merge back into both `develop` and `master`.

A hotfix branch is like a release branch, however it is used to create an emergency new release to fix an urgent bug in the production code that cannot wait until the next release.

A release branch is created by branching from `master`:

```
$ git checkout -b hotfix-VERSION master
update the version number
$ git commit -a -m "Updated the version number to VERSION"
```

Then, some commits are made to fix the bug, and it is merged into both `master` and `develop`:

```
$ git checkout master
$ git merge --no-ff hotfix-VERSION
$ git tag -a VERSION
$ git checkout develop
$ git merge --no-ff hotfix-VERSION
$ git branch -d hotfix-VERSION
```

2.2 Github Flow

GitHub Flow was designed for projects that deploy very frequently, perhaps even multiple times a day. As a result, it is a much simpler system that Git Flow (and may be less suitable for projects with a more fixed release schedule).

There are four “rules” in the system:

- Everything in `master` is deployable, and either is deployed or will be shortly.
- When working on something new, create a descriptively-named branch from `master`.
- Push your changes to the server constantly.
- Use GitHub's Pull Request system as a code review tool.

And so, say you wanted to make a feature named "foo", this is how you would do it:

1. `git checkout -b foo master`
2. Do some commits, pushing after every few commits to `origin/foo`.
3. Open a pull request to merge `foo` to `master`.
4. Make any changes which are discovered to be necessary in the resultant code review, until the code is ready.
5. Merge `foo` to `master` and push.
6. Deploy `master` to the production system ASAP.
7. `git branch -d foo`

3 Some Best Practices for Teamwork

When using VCS by yourself, you can be a bit sloppy in some ways: you can be behind on pushing commits, you can write bad commit messages, and so on: things that don't really work well when in a team. Here's a list of things I consider more important when working with others:

- Pick an indent style and stick to it.
- Don't make needless whitespace changes. The occasional commit to fix formatting is fine, but it should be a commit by itself, rather than a mixture of code changes and unrelated whitespace fixes.
- Write good commit messages, the Linux kernel guidelines recommend having a summary sentence, a blank line, and then a paragraph or so of text explaining what was done in detail. This may not be necessary for your project, but a good summary at least is a must.
- Pick a workflow and stick to it. Git Flow if you have a release schedule, GitHub Flow if you release all the time.
- Push commits frequently, you should never have more than two or three waiting in your local repo.
- Fetch commits frequently, certainly before you push.
- Communicate all the time: not really a VCS thing, but it is a must for any project with multiple developers. IRC, meetings, email, whatever.
- Each logical change should be its own commit.
- (Depending on workflow) commits should not result in a broken system.
- (Depending on workflow) have a good testsuite, and run it before every commit. Perhaps even have your VCS do this automatically and reject broken commits.

Really, all the problems that you may face when working in a team that would not have arisen if you were working individually can be solved with good communication and planning. If you get that right, everything will naturally fall into place.