Version Control and Teamwork

Michael Walker

30th January, 2013

Contents

1 Version Control **2**
1.1 Distributed vs. Centralised 3
1.2 Some popular VCS tools 4

2 Using Git **4**
2.1 Git Flow 4
2.2 Github Flow 4

3 Some Best Practices for Teamwork **4**

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.

Deploy 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 deploy 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 deploy 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

2 Using Git

2.1 Git Flow

2.2 Github Flow

3 Some Best Practices for Teamwork