Using git, the distributed version control system - Part 1

October 14, 2011

1 Introduction

Version Control systems (VCS) help us track changes to files over a period of time. As professional developers, you will almost always have to work with some kind of VCS during the development of any non-trivial project. This lesson provides an introduction to a powerful version control tool called *git*.

Git is a complex tool and it is easy for beginners (especially those without experience developing non-trivial projects) to get confused. In this lesson, I focus on the interesting things you can do with *git* without getting into the finer details of how each command works.

Screencasts:

1. gitbasics/gitbasics.mp4
2. github/github1.mp4
3. github/github2.mp4
4. github/github3.mp4

2 The nature of a real-world software project

If your only experience with programming is writing small toy programs in say C/C++, Python, Java (or any other language), you need to first get some insight into the nature of a real-world software project. Let's take the case of the Linux operating system kernel. An old version of the Linux kernel, version 2.6.26, has about 60 lakh lines of code (mostly C) in it[1]! This code is produced by hundreds of developers (statistics released in Feb 2007, relating to kernel version 2.6.20, mentions that over 2000 people had contributed code; the article is available here: http://lwn.net/Articles/222773/). The key ideas here are:

[1]This data was taken from here: http://linuxator.wordpress.com/2008/07/22/5-things-you-didnt-know-about-linux-kernel-code-metrics/

1. Real world code bases can be huge - you can expect projects containing many lakhs of lines of code.
2. All this code can't be stored in a single file - you can expect to see hundreds (or thousands) of source files.
3. The code evolves constantly as new features are added and bugs are fixed.
4. The code is not written by a single person - tens (or even hundreds of people) might collaborate actively on a single project.

Managing large, constantly evolving code bases with contributions from many different developers throws up a lot of challenges - version control tools like *git* are designed to address such challenges[2].

3 Understanding a small part of the problem

Let's get into the mind of a developer who is working on a new project from scratch (assume she is working on a C project).

On Day 1 of the project, our developer writes 300 lines of code split into three files: *a.c*, *b.c* and *c.c*.

On Day 2, she adds a new file *d.c* and makes some changes to file a.c created on Day 1. She postpones testing this code to Day 3.

On Day 3, she finds out that the changes to *a.c* she made on Day 2 has resulted in a bug; she wants to now see the contents of the file *a.c before* she had applied the changes on Day 2.

If she had created a backup for *a.c*, this would be an easy job. Otherwise, our developer is going to waste some time trying to figure out the changes she made to *a.c* which resulted in the introduction of a new bug.

This is a very common problem which is easy for even beginners to understand. If you don't have access to a version control tool, you might solve this issue by taking complete backups of your project every day. If you are starting work on say June 1st, you might think of creating a folder *june1* which holds all the files created on June 1st. Next day, you can create a folder *june2* which is a copy of *june1*. All the changes/additions you make on June 2nd will be done in the folder *june2*. This way, you always have access to previous versions of your project. If the changes you make on one particular day results in buggy code, you can always go back and get the bug-free version for comparison.

The problems with this approach are not very hard to spot. For one, this is really tedious. Imagine a project with a few million lines of code in it; the source code itself might be hundreds of mega bytes in size. Imagine creating

[2]This does not mean that you should use git to manage only large projects; it is a good idea to use a tool like git to systematically handle the source code of any non-trivial project which is under constant evolution. I use git to manage the LaTeX/LyX code from which I generate these course notes; the Java code for the recursive-labs.com web app is also managed by git.

complete backups of such a project every day! To compound the problem, what if you have ten developers in the team?

Version control tools offer a clean solution to this "travelling back in time" problem. Let's get started with *git* in the context of this problem.

4 What is git?

Git is a distributed[3] version control system originally developed by Linus Torvalds (the father of Linux). Many Free Software projects including the Linux operating system kernel use git as a version control tool.

5 Getting started

5.1 Installing git

You can type *git* at the command prompt and verify whether it is already installed. If not, you can install it by running (if you are using an Ubuntu system):

```
apt-get install git-core
```

5.2 Basic Configuration

Before you start using git, you need to give it two pieces of information: your name and email address. You can do this by running the following commands at the command prompt[4]:

```
git   config   –global   user.name    "YourName"
git   config   –global   user.email   "YourEmail"
```

Git will save the name and email id in a special file in your login directory called *.gitconfig.*

5.3 Creating a "repository"

Let's suppose you are a software developer starting out on a new project[5]. It is Day 1 of the project and you are going to write the first few lines of code.

Your first step will be to create a new directory (let's call it *proj*) to store all the files associated with the new project. So you execute the commands:

[3]There are two kinds of version control systems - distributed and centralised. A tool called *svn* is a good example of a centralized version control system while tools like *git* and *mercurial* are examples of distributed version control systems. We will not go into the details of the centralized vs distributed approaches to version control in this lecture. Interested students should check out: http://en.wikipedia.org/wiki/Distributed_revision_control

[4]Note that there are two hyphens in "–global"

[5]Keep in mind that this is just a "toy" project - the idea is to demonstrate the use of git.

```
mkdir   proj
cd   proj
```

Now, you want git to manage the files under this folder[6]. For this, you have to execute the following command[7]:

```
git   init
```

When you enter the above command, you will see git responding like this[8]:

```
Initialized empty Git repository in /home/recursive/proj/.git/
```

You are now ready to get started!

5.4 Day 1, adding the first file

You write your first few lines of code and save it in a file called *a.c*:

```
main()
{
   printf("hello,project\n");
}
```

You are ready to wind up your day's work; you feel that it is now time to *commit* your work to the safe custody of *git.* So you execute:

```
git  add   a.c
git  commit  -m  'First commit'
```

The magic of *commiting* your work to git will become evident when you come to work on Day 3; stay tuned![9]

5.5 Day 2, modifying a.c and adding a new file b.c

Your second day's work involves making a small change to *a.c* and adding a new file *b.c*.

Here is the modified *a.c*:

```
main()
{
   printf("hello,world\n");
}
```

And, here is the new file *b.c*:

[6]Technically speaking, we say that we wish to convert this folder into a git *repository.*

[7]Make sure that you are under the *proj* directory when you execute this command

[8]The "git init" command creates a directory called .git - this folder will contain some files and directories which form the basic framework used by git to track revisions.

[9]The -m option to "git commit" is used for specifying a commit message. We will soon see how this message is useful.

```
int sqr(int x)
{
   return x*x;
}
```

When you are finished with your work, you can *commit* your files by executing the following commands:

```
git add a.c
git add b.c
git commit -m 'Second commit'
```

5.6 Day 3, you wish to go back in time!

On Day 2, you made a change to the file *a.c*. Now, on Day 3, it so happens that you want to know what the file *a.c* looked like before you made the change on Day 2.

Here comes the magic; try this at the command prompt:

```
git show master~1:a.c
```

And, you see the content of the file *a.c* exactly as you had made it on Day 1! Note that your project directory contains only one copy of the file *a.c* - git works its magic by creating a folder called .git (which you will see only when you execute the *ls* command with the *-a* option) and storing revisions of every file that you create/alter in data structures situated under this folder.

The "git show" command shows you the content of the file *a.c* the way it was present in the just previous commit (that is what the *master~1* notation implies). If you wish to see the content of *a.c* two commits back, you can use the notation *master~2*, and so on.

Here is another experiment; execute the following command:

```
git log
```

You will see an output which looks similar to the following:

```
commit 72103518d4434f55085b5fbe723e10be2a91ce1c
Author: Pramode C.E <mail@pramode.net>
Date: Thu Aug 11 22:25:54 2011 +0530

Second commit

commit 5e14b90d73092bf2e321ce97b38636f21704f652
Author: Pramode C.E <mail@pramode.net>
Date: Thu Aug 11 22:15:47 2011 +0530

First commit
```

You are seeing the *commit messages* (the messages you supplied using the -m option to *git commit*) as well as your name and email id (which you added at the very beginning using *git config*). The *git log* command is useful when it comes to seeing the evolution of your project over a period of time. In a real project, the commit messages should be something meaningful like: “fixed a bug in the memory allocator”.

5.7 Branching and merging

Say you have worked hard for one month and reached a stage where your project is stable and can be shipped to the customer. Things become more complex at this point. You will have to keep on enchancing your product, but until those enhancements are all properly integrated and thoroughly tested, you still need a copy of the *stable* code. Your customers may some time discover bugs in your code - you might then have to go back to the stable version and fix the bug. Your development has now become non-linear. You need to have a *branch* where you have *stable* code and another branch where you have *experimental* code. You will often have to switch back and forth between these branches - adding/deleting files and making modifications here and there[10].

You may argue that this can be easily done by taking a copy of the stable code and making changes to that copy. But this will soon turn out to be complicated. For example, say that after two more months of development and testing (in the experimental branch), you are ready to announce another stable release of your product incorporating all the work done in the past two months in the experimental branch. You will definitely wish to include all the bug fixes made in the stable branch in this new release - you sort of wish to *merge* the stable and experimental branches.

Git shines in managing such branches-and-merges effortlessly. Let’s find out how.

5.8 Day 3, you want to start a branch

After two day’s of development and testing, you are planning to start a new branch. All the code that you write on Day 3 will go into that branch (called say *experimental*).

Here is how you start a new branch:

```
git checkout -b experimental
```

Git will respond with a message:

```
Switched to a new branch 'experimental'
```

You can now create a new file; c.c:

[10]You may also think of two different ways to implement a new feature - you want to try out both and see how they perform. It will be great if you can work on two different branches and keep these modifications separate.

```
int cube(int x)
{
    return x*x;
}
```

Finally, you can commit the file by executing the commands:

```
git add c.c
git commit -m 'first commit on new branch'
```

By the way, you can try out the command: *git branch* to view the branches you have created. You will see two names in the output - *master* and *experimental.* The name *master* refers to the default branch you are on when you initialize a new git repository. The "*" symbol before the name *experimental* indicates that you are currently on that branch.

5.9 Day 4, add one more file

You are going to add one more file on Day 4 (called *d.c*):

```
int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

Now, add and commit this file:

```
git add d.c
git commit -m 'second commit on new branch'
```

You have now added two files to the *experimental* branch, *c.c* and *d.c.* You are currently on the new branch called *experimental*; an *ls* command will display the names of four files: *a.c*, *b.c*, *c.c* and *d.c.*

5.10 Day 5, you are planning to work on the old branch (called "master")

Here comes the magic! On Day 5, you wish to switch back to the old *master* branch and continue working there. So you do:

```
git checkout master
```

Git will respond with the message:

```
Switched to branch 'master'
```

If you execute the *ls* command, you will see only two files in the current directory: *a.c* and *b.c.* Why? Because you added *c.c* and *d.c* to the branch *experimental* and now you are back on the old, *master* branch which holds only *a.c* and *b.c*. You can execute *git branch* to verify that you are back to the *master* branch.

Suppose you wish to continue working on the master branch. You can add one more file, say e.c:

```
void swap(int *a, int *b)
{
    int t;
    t = *a, *a = *b, *b = t;
}
```

You can add and commit this file:

```
git add e.c
git commit -m 'Implemented swap function'
```

Now, the *master* branch holds an additional file *e.c*.

5.11 Day 6, merging "master" and "experimental"

You have finished your work on the experimental branch; now you wish to *merge experimental* with *master*. In a real world scenario, you have tested your experimental code thoroughly and you are planning to release a new version of your product which incorporates all the innovations developed in the last few months with the bug fixes made to the stable code base.

In git, it is as simple as executing:

```
git merge experimental
```

Git will combine the modifications made in the two branches into the *master* branch. If you do an *ls*, you will see the files *a.c*, *b.c*, *c.c*, *d.c* and *e.c!*

You may now delete the experimental branch by executing:

```
git branch -d experimental
```

You can verify that the branch has been deleted by running:

```
git branch
```

6 Conclusion

We have completed a whirlwind tour of some of the interesting things which *git* can do for us. Part 2 of this lesson will demonstrate some more fun stuff!