Using Git to Manage Source RTL

CS250 Tutorial 1 (Version 082311)
August 24, 2011
Brian Zimmer

How to use this tutorial

This class will be using Git for all of the labs and projects. This will allow the staff to easily view and grade your files, enable version control and backup for more efficient development, and make collaboration easier for the team project.

Plenty of information is already available about Git, and the best sources are listed below. For the labs, basic knowledge of Git is assumed. In this tutorial, a very basic introduction is given and common commands are listed, but it is highly recommended that you go over some of the references listed below (which vary in length from few page introductions to entire books).

Resources

- `http://www.youtube.com/watch?v=OFkgSjRnay4` Git in 1 Hour: A great video introduction to Git.
- `http://git-scm.com/documentation` Official Homepage: has links to all of the best places to learn about git, including free books.
- `https://git.wiki.kernel.org/index.php/GitFaq` FAQ

Why Version Control?

Version control systems take a "snapshot" of what your files look like at a certain point in time. This is an excellent way to keep a backup of your files and easily see how things have changed over time. Also, version control allows for multiple people to work on the same code. Major software programs with hundreds of developers use version control systems to merge everyone's contributions. Even though there is a learning curve, version control systems save time in the long run. Making copies of files for backups or emailing files between developers might work, but version control provides much more efficient and easy ways to perform common tasks.

Why Git?

Git is different from other version control systems in that there is the concept of both global and local repositories. In a traditional system, such as Subversion, every time you want to "commit" (remember) the state of your code, it is sent to a shared repository. The problem with this is that other people can see all of your commits. Many times what you are working on will break the overall system. If you commit regularly, you will break the code for others. But in Git, you have a separate local repository. You can commit as much as you want, and only share your changes with

others once they are more mature. This also means operations are much faster, as everything is usually happening locally.

Example workflow

We have created a sample repository for you to experiment with.

```
% git clone ~cs250/git/git-tut.git/
% cd git-tut
```

Start by creating an example file. Add any text you would like to it, and feel free to substitute vim with your editor of choice.

```
% vim TODO
% type type type...
```

Now lets see what Git sees.

```
% git status
```

Git views this file as an untracked file. This means that Git is ignoring the file for now and will not know about any changes that happen to this file.

```
% git add TODO
```

Now check what Git sees by running `git status` again. Git now says that it is tracking this file. But you can still change this file, and Git won't remember what you did to it. Right now, the file is only "staged". For Git to remember what the file looked like at a certain point of time, you need to commit it (take a snapshot of it).

```
% git commit -m "log message"
```

If you don't include the `-m "log message"` option, a text editor will open. Enter the log message there, save, and quit, and then the commit will happen. Run `git status` again to see that Git believes everything is up to date as expected.

Now lets see how we view history. Open your file, and add another line.

```
% vim TODO
% type type type...
```

To see what has changed, run:

```
% git diff TODO
```

Now commit these changes as well:

```
% git add .
% git commit -m "log message"
```

Where `git add .` will add every file in the current directory. Note, the shortcut to add and commit in one step is:

```
% git add .
% git commit -am "log message"
```

Now add another line again:

```
% vim TODO
```

Imagine you didn't like your changes, and wanted to revert them:

```
% git checkout -- TODO
```

The `--` exists because `git checkout branchname` will open a branch (a more advanced topic) and the `--` lets Git know that you only want to checkout a file.

Another useful feature is that you can view previous versions of a file. Imagine a bug got introduced, and you would like to see a previous version of a file.

```
% git log
```

To graphically view this log, try the following.

```
% gitk
```

Notice next to the words "commit " there is a string of letters and numbers called a SHA. This uniquely identifies the commit. You can find these both with `gitk` and `git log`. To view the version of TODO for a given commit, paste this number below (yours will be different)

```
% git show 90940d1cdbb016952db2ba368fdc4906e010a9ff:TODO
```

Notice after the `:` is the path to the file. In general, you don't need the full SHA. Also, if it is a long file, you might want to pipe it into your favorite editor.

```
% git show 90940d:TODO | vim -
```

But what if you had dozens of files, and you wanted to see the entire state of the repository from an earlier commit? Using the SHA found from `git log` or `gitk`, just "checkout" this revision.

```
% git checkout 33c9894e13f8c3d9d4307c4b77e1a92e286d70ba
```

Once you are done and want to return to your old state:

```
% git checkout master
```

Make sure you commit before going back through history so you don't lose any files.

Last, if you are working with others, you can push your changes up to the remote repository. Whenever you want to do this, you need to ensure two things. First, make sure that you have commited all of your local changes and your `git status` is clean, otherwise you will run into problems. Second, make sure your local copy is up to date by pulling changes from the remote repository.

```
% git pull origin master
```

`origin` refers to the default remote repository (you can have multiple remote repositories) and `master` refers to the fact that you want to checkout the main flow (and not a branch). These can usually be ignored, and `git pull` will work fine.

Read the status of the pull. If the remote version change since the last time, it will attempt to merge the changes automatically! Usually, this magically works. For example, if a new file was added, it will just download the file. If a line was added to a file that you changed as well and the lines are far apart, both changes will be merged together. But if the same file was changed too much either remotely or locally and Git cannot figure out how to merge them, you will see an error:

```
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

To fix this, edit the files that it listed (in this case, README) Portions where problems occured are marked inside the file. For example:

```
<<<<<<< HEAD:README
Example empty git repository
=======
Example of an empty Git repository
>>>>>>> f12702bfd22e037cf55d0e7e2da4e9f0f1d58f62:README
```

The version on your local file system is on the top and version that the remote repository has is on the bottom. Fix the merges by choosing one of the two and deleting the markers.

You can also use a tool to do this by typing `git mergetool`. Try entering vimdiff as the merge tool, and you will see the 3 panes. The left holds your file, the middle the file to be resolved, and the right holds the remote file. Modify the middle pane until conflicts are gone.

Once merges are fixed, commit your changes (if you use the mergetool, it will automatically be committed when you quit, and a filename.orig file will hold a backup of the problem file)

```
% git commit -m "fixed merges"
```

Now that everything has been merged, you can push the changes back to the remote repository.

```
% git push origin master
```

Ignoring files

Files that are dynamically generated should not be stored in repositories. Keeping them in a repository is redundant (as can be generated from other files in the repository) and can lead to a mess.

"make clean" is commonly used to delete these dynamically generated files. In git, we will create a file that will ignore these same files.

In the top directory, run

```
% vim .gitignore
```

Then inside this file, list directories or files that should not be seen by the repository individually on each line. Wildcards are acceptable.

```
build*
*.log
tmp/
```

Then running `git status` should no longer show these files.

Advanced topics

There are many more features to Git, and only the bare minimum has been introduced. A few other features are worth learning about (especially if multiple people share a repository)

- `git blame`: Will show who change every line on a file and when
- `git bisect`: Can go back through previous revisions using a binary search and determine when a bug occured
- Branching: A different method of working where each feature has its own branch and the master branch is kept cleaner. Great for experimenting.
- Editor plugins: Emacs and Vim both have excellent plugins that allow for Git commands to be performed from within the editor

Cheatsheet

`git clone (url)`	Pulls a repository from a remote server and creates a local version
`git status`	See what files have changed since the last commit
`git diff`	See what has changed
`git add (filename)`	Stage a file for the next commit
`git commit -m "Message"`	Commits staged files
`git pull`	Update from remote repository
`git push`	Push up latest commit to the remote repository
`git checkout -- (filename)`	Reverts a file to the last commited version
`git clean -dfx (dirname)`	Reverts an entire directory to what it looks like in the repo
`gitk`	Graphically see commit history