Git in a Nutshell

for normal people (version 0.1)

Jonas Jusélius

<jonas.juselius@chem.uit.no>
University of Tromsø
Department of Chemistry
N-9037 University of Tromsø, Norway

Rev.1:
`git clone http://git.jonas.iki.fi/CVS2git.git`
Corrections and improvements are most welcome via e-mail using the `git-format-patch` *facility.*

1 Introduction to git

Git is different from CVS in most respects. When you clone a repository you get the WHOLE repository, everything, not just a working copy like in CVS! `git-clone` is almost the same as doing a

```
$ scp -r myserver:/path/to/myrepo .
```

Within this personal repository you can do *whatever* you like, i.e. create branches, delete branches, tags, and of course commit as much as you ever like. It's only when you push to the master that others can see your changes.

2 Branches in git

2.1 Remote branches

The thing that differs most from CVS when working with git is the use of branches. Under git branches are used extensively! When you clone a repository (this is called the 'remote' repository, and is referred to as the 'origin') all the branches in that repo are renamed in your local copy by prefixing the branch name with '`origin/`'. These branches are called 'remote branches' or 'tracking branches', to distinguish them from *your*local branches. `git branch -r` show all remote branches. When you run `git-fetch origin` *all* remote branches are updated to the exact state of the remote repository. Now you can inspect the changes on those branches by either using `gitk`, or a combination of `git-log`, `git-whatchanged`, `git-diff` and `git-annotate`, e.g. to inspect changes on the most important remote branch: `gitk origin/master`. Note that `git-fetch` updates the remote tracking branches, not your working branches! Never, ever, checkout a remote branch directly unless you absolutely want trouble.

2.2 Tracking remote branches

In order to incorporate changes in in remote branches you need to merge your working copy with the remote branch, e.g. (`git-checkout mybranch`), `git-merge origin/master`. If you get a conflict, run `git-mergetool` which will fire up the merge tool of your choice. When you have resolved all issues, run `git-commit` to complete the merge. The `git-pull` command basically does a `git-fetch` and `git-merge` in one go. I recommend you *do not* use it unless you very carefully read the man page first, since it has some pitfalls!

As I mentioned earlier, never work on a remote branch directly! This will cause terrible problems when you try to sync with the remote repository. Instead create a local branch *from* the remote branch and do your work on it instead: `git-branch foobar origin/foobar`; `git-checkout foobar`. After a `git-fetch` run `git-merge origin/foobar` to get the latest updates on the branch (and also `git-merge origin/master` if you want to track the "official" changes). When you feel you have something worth showing to others (or just for backup) you can push your changes on some branch to your personal branch on the master server, i.e. the remote repository, `git-push origin mybranch:myuserid`.

2.3 Daily git

All this might sound a bit confusing, so here are my recommendations for how to work with git until you get used to it: When you clone the master repo, you will get a set of remote branches (`git-branch -r`), and one local working branch called 'master'. This branch is in the exact state of the remote master branch (i.e. origin/master and master are identical). I suggest you keep it this way, and create a working branch for yourself, e.g.

```
$ git-branch work
$ git-checkout work
```

Now work like you normally do, and every now and then (every morning for example) do

```
$ git-fetch origin
$ gitk origin/master
# if you like what you see
$ git-merge origin/master
```

Every time you have made changes which can be considered complete in some sense (you know best), do a commit (it's much better with many small commits than a few big commits as you will see):

```
$ git-status
$ git-add file(s)
$ git-commit
```

3 Cherry picking

That's really it! Of course there are still some nice tricks we can play with branches. Suppose you want to try out some idea, but you don't know exactly how it will work out. Create a new branch from your current working branch and check it out:

```
$ git-branch foobar
$ git-checkout foobar
```

Now work and commit often. If it turns out that everything is good and you want to keep all changes, merge them with your working branch and push them to the master:

```
$ git-checkout work
$ git-merge foobar
$ git-push origin work:myuserid
```

Now you can delete the foobar branch for ever and all times:

```
$ git-branch -d foobar
```

If, on the other hand it turn's out that you had a crackpot idea, and you don't want to see any of it anymore, delete the branch and all changes, commits and everything on the branch will be gone for ever! No trace of it. But what if there are partially useful changes to foobar that you want to keep, before discarding the rest? Well that's when small commits are useful because you can cherry pick! Chekout your work branch, fire up gitk on foobar, and find the commits you like to keep. Every commmit is identified by a long SHA1 hash (a long sequence of numbers and letters). Now cherry pick the commits you want into work:

```
$ git-checkout work
$ gitk foobar &
# find commit, copy the SHA1 hash with the mouse
$ git-cherry-pick SHA1
```

Repeat the cherry pick as many times you like. As you see, git gives a lot of flexibility by using branches. Just be a bit careful in the beginning not to make a mess and lose track of what you are doing.