Git isn't that Hard

Jan. 25, 2017

Here are a slowly growing summary of Git settings and commands I find useful, with short explanations following, addressed to the reader in second person.

I claim no intention to make it a complete table, even in the most rudimentary sense. What I have supposed to be helpful, I summary and explain here, and what I have not, I do not. It has not been long since I started learning to use Git, and many point are simply paraphrased from some Stack Overflow answer. Thus, I deeply appreciate the reader to remind me of any error.

I use Mac OS (presently Sierra) on MacBook Air. From now on, by convention, commands meant to be run in Bash shell is prefixed with a `$` for clarity. Mac users new to shell concepts may just open the Terminal.app, copy and paste whatever I quoted (without `$`), and hit the Return key.

Git commands require the current working directory set to be the top directory or any subdirectory of it except except those inside `.git/`. There are workarounds if you really don't want to do this. But I guess it is safest just to always go (by `cd`) into the top directory. This will be assume true hereafter.

Before we continue, keep in mind that Git comes with a detailed manual. If you have any question of `fetch`, for example, open Git man pages by:

```
$ git help fetch
```

Or equivalently,

```
$ git fetch --help
```

Installing Git

- In Mac, Xcode is built-in. Xcode includes Command Line Tools, which in turn includes Git. In case you need to install Xcode, find it in App Store.

However, Xcode takes up a lot of space, so if you don't develop software for Apple, you may want to uninstall it. You can still install Command Line Tools for Xcode only even if you don't install Xcode. To do this,

```
$ xcode-select --install
```

- You may also directly install Git using homebrew:

  $ brew install git

Checking Installation of Git

To check current version, so that you know whether update is successful:

git –version

Mac user may not be allowed to overwrite the built-in binary that came with Xcode. If you are determined to overwrite,

```
$ brew link --overwrite git
```

To make sure this is effective, you may check current location of binary:

```
$ which git
```

if the result is `/usr/bin`, then this `git` is the built-in one. If it is `/usr/local/bin`, this `git` is the user-installed.

Basic Concepts

It is best that the reader take some time and understanding the underlying mechanism of Git, so that we can better describe its operations and grasp their meaning.

- A *tree* is an aggregate of nodes, each of which contains a pointer of its own child or children, if any.
- A *tree object* is a representation of directory. Each subdirectory of its is represented by another tree that it points to, and each file of its by a blob that it points to.
- A *working tree* is the totality of actual source files you are working on. All you see and edit (using Git or not) belongs to the working tree. For convenience, I say that there is a *top directory* which is the minimal directory that encompasses the working tree.
- A *local repository* (local repo) is a subdirectory in the working tree named `.git/`. Everything Git needs is saved inside the
- An *index* lists files in the working tree Git is aware of. The index is saved in the local repo. We say that Git *tracks* these indexed files.
- A *blob* is an opaque binary file that records the the content of a file in the manner Git may read.
- When you *commit*, you create a *commit object* that include sufficient information to describe the present working tree, a unique *hash* generated according to author, time stamp, file contents and other figures.
- Commits are linked to each other according to respective relation, which makes up a *commit history*.
- A *stash* is the collection of blobs not yet committed.
- When you *clone* a repo, you you create a copy of it.
- When you *fork* a repo, you clone it and modify it, thus generating two diverging versions, and subsequently different histories.
- A *remote repository* (remote repo) plays a similar role to local repo, but is store in an Git hosting service, like GitHub. It stores a working tree and commit history.
- When you *push*, you download files from the remote repo to to those of local repo.
- When you *pull*, you upload files from the local repo to those of remote repo.

Creating Repos

Creating Local Repo

Now let us say that there is a directory called `~/templates_configs_notes`. Run `cd` into it:

```
$ cd ~/templates_configs_notes
```

and generate local repo with:

```
$ git init
```

Following our example, the local repo will be named `~/templates_configs_notes/.git/`. Remember that nothing affects the remote repo unless you push. The implication is that if you have screwed up something and want to start anew from the very beginning, willing to abandon the not-yet-pushed changes in the working tree, you can simply delete `./.git/` and `git init` again.

Creating Remote Repo

To create a new remote repo, sign up a Git service provider if you haven't. No wonder the most popular choice is GitHub. Bitbucket is also widely used. The greatest difference relevant to average user is probably that Bitbucket offers unlimited private repositories for free while GitHub does not, but that Bitbucket limits number of users to 5 for free while GitHub does not. In short, sign up both.

The following instructions apply to GitHub, but the main idea is similar. In your personal homepage, click the plus sign on the upper right corner, and choose "New repository". There will be the first branch by default, which is always called `master`.

When you name your remote repo, note that GitHub disallow special characters, and forbids even underscores, but allows hyphen. I deem it a good convention to name local repo with underscore-separated words, and remote repo with the same name with hyphen-separated words. Traditionally repos are named all in lower case. Following the above example, let us call it `templates-configs-notes`.

GitHub offers two protocals: `https` and `ssh`. Enter the page for the repo, and click the green button "Clone or download". If the small title reads "Clone with HTTPS", there is a url that looks like

```
https://github.com/aminopterin/templates_configs_notes.git
```

Click "Use SSH", and you will see something like

```
git@github.com:aminopterin/templates_configs_notes.git
```

To set remote url, go back to your terminal emulator. If you use SSH Key, set

```
$ git remote set-url origin git@github.com:aminopterin/templates_configs_notes.git
```

In other words, the remote url is abbreviated as `origin`.

To list currently existent remote repo(s):

```
$ git remote -v
```

The result will look like:

```
origin  git@github.com:aminopterin/templates_configs_notes.git (fetch)
origin  git@github.com:aminopterin/templates_configs_notes.git (push)
```

To Commit and Push

To Compare

Before committing, it is helpful to see what new works was done. To show modification not added to the index, that is, differences between working tree and the index:

```
$ git diff
```

If new work has been added to the index, you can still compare the index with latest commit in the local repo,

```
$ git diff --staged
```

To compare a specific file with the commited version in the local repo, say `README.md`,

```
$ git diff HEAD README.md
```

where `HEAD` is a pointer to the latest commit.

After any of these, you will see a new page that summarizes modified lines, and what is changed.

Navigation commands are identical to `less`, and similar to Vim. Hit `j`, `e` or `^e` to forward one line (where `^` stands for Control key). Hit `k`, `y`, or `^y` to backward one line. Hit `f` or `^f` to forward one window. Hit `b` or `^b` to backward one window. Hit `d` or `^d` to forward one-half window. Hit `u` or `^u` to backward one-half window. Hit `q` or `:q` to quit.

Updating the Index

If there are new files created or old files deleted, run this to update the index according to the present working tree:

```
$ git add -A
```

This does almost the same, except in that it ignores newly created folders:

```
$ git add .
```

To Commit

Before you commit, you may want to see a short summary of what files are changed and deleted, with

```
$ git status
```

To commit, use `git commit`. However a commit message is strongly recommended, for example,

```
$ git commit -m "Rename files using new convention"
```

Summarize your work concisely. If you cannot do this, you should have probably split your work to be two or more commits. It is the tradition that, to save space, verbs in base form are used, and a there is no period in the end.

You can modify you commit message even after you commit, if you run

```
$ git commit --amend
```

Now, an editor opens, showing the commit message in the top, where you may revise it. You can add more explanatory lines below. The status is shown again as commented lines. This file is saved as `./.git/COMMIT_EDITMSG`.

Afterwards, you can view the commit log with

```
$ git log
```

The result shows commit IDs, authors, time stamps, and commit messages. To make the log more concise and informative,

```
$ git log --all --decorate --oneline --graph
```

The option names are pretty explanatory. Mnemonic: “a dog”.

Branching and Manipulation

To Clone and Fork

To Create New Branch

To Mergjk

Configuring Git

Keep in mind that it is the same thing to run, in the terminal,

```
$ git config --global foo.bar quux
```

as to append the line

```
[foo]
    bar = quux
```

to `~/.gitconfig` (the second line should start with a tab). I will use the former command in the following.

Ignoring Binary Files

It is not in general Git's purpose to track binary files. Binary files cannot be compared by utility `diff`, thus cannot be merged or rebased in the normal sense. So you don't track binary files unless it's really needed by some routine and necessary to be upload it to the remote repo. For example, a logo of your program that you want those who clone to see.

To tell Git to ignore some specific binary files, all files in some folders, or all files having a certain extension, create a file named `.gitignore` in the top directory. If, for sake of illustration, your `.gitignore` has these lines,

```
.DS_Store
*.pdf
sketch/
```

Then the `.DS_Store` file in the top directory, all `.pdf` files anywhere, and all files in the folder called `sketch/` in the top directory, will be ignored by Git. You can play around with wildcard `*`.

Specifying Text Editor for Commit Message

The default editor for commit message is vi. However, some of the lines in my `.vimrc` is not compatible with vi (which is normal), and Vim is used anyway with wrong color settings.

To set Vim as the commit message editor, put these in `~/.bashrc`:

```
export VISUAL=vim
export EDITOR="$VISUAL"
```

You can set it to be your favorite editor.

Alternatively, you can set the editor in Git's part, with

```
$ git config --global core.editor "vim"
```

Color

To color Git's console ouput,

```
$ git config --global color.ui auto
```

All the various `color.*` (where `*` is wildcard) configurations available with git commands, will then be set. The `auto` makes sure Git try only those colors the current terminals emulator supports. When a command is used for the first time, relevent color configuration will then be set.

Showing non-ASCII characters

By default, non-ASCII characters, such as Chinese characters, are backslash-escaped according to C-styled character codes. For example, "我" becomes `\346\210\221`. To show them as verbatim,

```
$ git config --global core.quotepath false
```

More Information

- The Git Documentation

  This is the documentation on the official homepage of Git. Same material may be found in `man` pages that are included in the Git package itself, to quote the site.

- Richard E. Silverman (2013). *Git Pocket Guide*. Sebastopol, CA: O'Reilly Media.

  Guides for Git are abundant. This is a short guide that may both be read from cover to cover, and looked up as reference.

- Git—The Simple Guide

  A table of for the most common Git commands, and very brief explanations. On the site, there is a downloadable PDF version.

- Stack Overflow

  Stack Overflow is still the most likely place you end up with if you google your problem, but, since everyone can submit, you should take their advice with a grain of salt.

Alright. Maybe Git really is *that* hard.