A Short Introduction to Git

Zhu Guangyu 20165953

September 19, 2018

1 What Is Git?

Git may be the most popular distributed version control system in the world. It is created by Linus Torvalds for development of the Linux kernel. Git is primarily used for source code management in software development, but it can be used to for any set of files.

Next, let's briefly talk about the Version Control System and the features of Git briefly.

1.1 Version Control System

How do you manage your files? Do you do things like this when you try to modify your files?

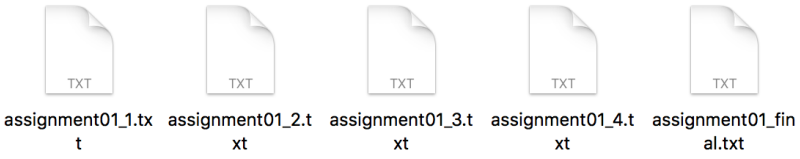

Figure 1: "Old-fashion Version Control

With version control, you do not need to make duplicate files to avoid screw your files up anymore. Version control system is a system that records changes to files over time so that you can go back to the previous states. Also it provides more features, such as comparing the differences between two versions and collaborating with others.

1.2 Git

In general there are two types of version control systems, *Centralized Version Control Systems* and *Distributed Version Control Systems*. Centralized system has a client-server structure. All files are stored in a central server and users just can access specific files which administrators allow them to.

Like we said before, Git is a *distributed version control system*. Instead of certain files, it fully mirror the repository from the server, including history. This approach gives distributed system some advantages that centralized system does not have. Such as users can work offline; if one server dies, any client can restore it easily. Furthermore, we can have several remote repositories working simultaneously within the same project.

2 How to Use Git

At this part, let's talk about how to use git. I will just list some basic operations to let you get into Git's world. If you want to know more, Pro Git is a good book to read. You can get it free online.

2.1 The Baic Construct of Git

The files in git has three main states: committed, modified and staged:

- Modified means that you have changed something but have not put them into your database yet.
- Staged means that you tell git which modified files you want to commit.
- Committed files are files that stored in our local database

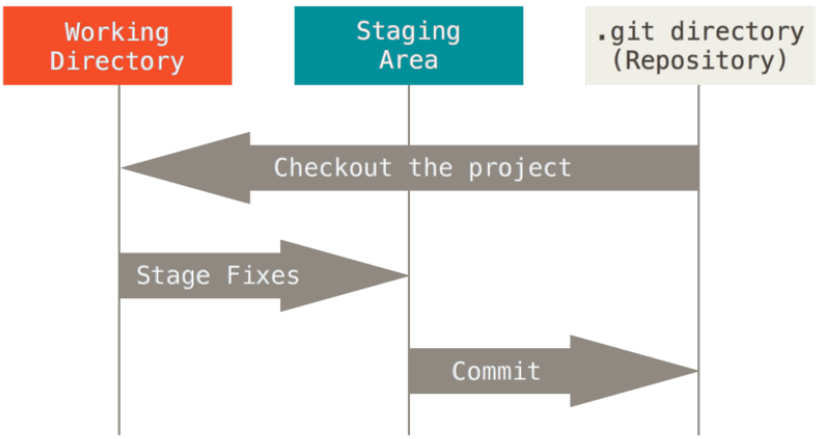

Figure 2: Git Stages

At the root directory of each git repository, you can find a hidden directory called **.git**. This directory stores all the data for your project. When you clone a repository from another computer, this is what you get. Delete .git, you will lose everything.

Another thing I want to talk about is *Branch*. One of the features of git is that it supports non-linear developments well. Branching means you diverge from the main line of development and to do some other works without

messing with main line. Git's branching operation's overhead is small. It does not copy the files but use pointers to link your project snapshots together. This approach makes creating branches fast and lightweight. That's why git encourages people to do branch and merge often.

We will talk how to use branch later.

2.2 First Time with Git

2.2.1 Set User Name and Email Address

After the installation, the first thing to do is to set your name and email address. This is important because Git uses this information to identify users. Setting by command `git config`:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "Your Email address"
```

Git's configuration files be stored in three different places and have different scope. Use options below to 'git config' to choose the scope you want. Each level overrides the previous level:

```
--system # apply to every user on the system
--global # apply to you, the user's repository
--local # apply to that single repository (default option)
```

There are many other variables we can modify. Check them once you got time.

2.2.2 First Repository

Initial Repository

Let's do our first commit together. Go to your project's directory, type

```
git init
```

This command will generate the **.git** subdirectory under the current directory.

At this point, nothing in the project is tracked yet. You have to tell Git which files it should working on.

We can check the status of files through command `git status`:

```
$ git status

On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Track Files

To track new files, use command `git add <file>`. This command will put files into the *staging area*, both new files or modified files. Only files in the staging area will be committed into the database. Now, let's add some files:

```
$ git add README.md assignment01.tex
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

  new file:   README.md
  new file:   assignment01.tex
```

We can find the two files are staged and will to into our next commit.

Commit

Now let's finish our first commit, just type `git commit`. Git will ask you to input the commit message:

```

  # Please enter the commit message for your changes. Lines
  starting
# with '#' will be ignored, and an empty message aborts the
  commit.
#
# On branch master
#
# Initial commit
#
```

```
 9 # Changes to be committed:
10 #   new file:   README.md
11 #   new file:   assignment01.tex
12 #
```

Writing meaningful commit message is very important, because the message is the best way to learn about a change. Google "How to write a good git commit", you commit message, but well formed message helps a lot. can find lots of articles talking about it. There is no standard way to write

Check History

Use `git log` we can check the existing commit history:

```
commit 7af35596163c980d0661faeef9a3eef1e67732be (HEAD -> master)
Author: Guangyu Zhu <guangyuzhu1129@gmail.com>
Date:   Mon Sep 17 22:04:38 2018 +0900

    Third commit

commit 38697bada55cf777d069613760f90a24dd09b792
Author: Guangyu Zhu <guangyuzhu1129@gmail.com>
Date:   Mon Sep 17 22:03:49 2018 +0900

    Second commit

commit b2e78948dde81c0ca5858e4f78d52a4b0676c368
Author: Guangyu Zhu <guangyuzhu1129@gmail.com>
Date:   Mon Sep 17 21:35:30 2018 +0900

    Initial the project.
```

There are hash values for each commit. Git use the hash values to identify commits. By the way, this is a example of bad commit messages. You can get nothing meaningful from above commit messages.

With these hash values we can go back to previous commit through command `git checkout <hash>`:

```
$ git checkout b2e78948
Note: checking out 'b2e78948'.

You are in 'detached HEAD' state. You can look around, make
   experimental
changes and commit them, and you can discard any commits you
   make in this
```

```
state without impacting any branches by performing another
    checkout.

If you want to create a new branch to retain commits you create,
     you may
do so (now or later) by using -b with the checkout command again
    . Example:

  git checkout -b <new-branch-name>

HEAD is now at b2e7894... Initial the project.
```

To go back to the newest commit, type `git checkout master`.

2.2.3 Delete and Ignore Files

Sometimes you tracked the wrong files or you want to remove some tracked files. Delete them from the directory may not working well, because you need to tell git to stop tracking them. Use `git rm <file>` will remove the file from git and working directory.

To avoid tracking some files that we do not need, **.gitignore** comes out. List patterns to match file names in **.gitignore**. Git will ignore all the files that are listed in **.gitignore** file.Git use *glob patterns* that shells use. If you do not know which files need to be ignored. Do not worry, gitignore.io can generate these rules for you. The only thing you need to do is input the working environment.

2.3 Branch

It's time for branch now. Like we said before, git use pointer to link snapshots(commits) together. That means we can use pointer to switch between snapshots easily. Creating a new branch just creates a new pointer for us to move around.

Create Branch

To create a new branch, use command `git branch <branch_name>`:

```
$ git branch testing
$ git branch
* master
  testing
```

`git branch` command without any option will show us current branches. The branch with * before it is the branch that we are working on.

Switch Between Branches

Switch to other branch by `git checkout <branch_name>` command. We use same command to check previous commits. Since they are just snapshots linked together. No actual difference between the normal commits and branches:

```
$ git check out testing
Switched to branch 'testing'
```

Now you can do anything you want with your files. There is no impact on the content on the master branch.

Merge

Assuming that we want to develop some new features. Here is the example of branch-merge workflow.

First we go back to master branch and create another branch to start development:

```
$ git checkout master
Switched to branch 'master'
$ git branch -b new_feature
$ Switched to branch 'new_feature'
```

`git checkout -b <branch>` will create a new branch then move on it automatically.

After the development, we merge the `new_feature` branch back into `master` branch to deploy the feature. We do this with the `git merge` command:

```
Updating 7af3559..2628c91
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
$ git checkout master
$ git merge new_feature

$ git branch
* master
  new_feature
  testing
```

Because we no longer need `new_feature` branch, we can delete it with `git branch -d <branch>`:

```
$ git branch -d new_feature
Deleted branch new_feature (was 2628c91).
```

Conflict

Let's do it again. This time we merge testing branch into master:

```
$ git merge testing
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result
    .
```

This is time something new occurs: **CONFLICT (content): Merge conflict in README.md**

What is conflict? Sometimes we modified same file on different branch, when we merge two branches together, git cannot know which one is the file that we want. Thus, we have to resolve those conflicts manually. Check status first:

```
$ git status
On branch master
You have unmerged paths.
    (fix conflicts and run "git commit")
    (use "git merge --abort" to abort the merge)

Unmerged paths:
    (use "git add <file>..." to mark resolution)

    both modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a
    ")
```

The message tells us both branches modified `README.md` file. Use editor open README.md:

```
<<<<<<< HEAD
Made some modification on branch new_feature
=======
Do some modification in branch testing
>>>>>>> testing
```

The content between <<<<<<< and >>>>>>> is the conflict part. Top block is the content in current branch(in our case is master). Above block is the content from incoming branch.

After we resolve the conflicts, stage the conflict file then commit the change:

```
$ git add README.md
$ git commit

$ git status
On branch master
nothing to commit, working tree clean
```

Finished the merge! In this context, we have tried some of the most basic and most commonly used commands. Next part we will talk about Github and how to use Github as our remote server.