Git 101 for single person and team

Yang Gao
July 3, 2014
yanggao1119@gmail.com

1 Forerunner

I have always wanted a git tutorial for beginners, that covers just the basics for single person and team. I found the official "GitHub Help" too simple, and many other tutorials too complicated. So I compiled my own.

2 Basics

2.1 What is git

See this plain intro[1].

The most essential concept in git is **branching**. Forking is a special branching between different repo owners. This is the git principle: **"Anything in the master branch is always deployable"**.

2.2 Why git

- Version control with branching
- Group collaboration with forking and branching
- Sharing files, group wiki, etc.

2.3 Suggested practice

- Always use command line in a terminal: you'll need to type a lot of "git" in your terminal;
- Always do your dev in a new branch so that you don't mess up your stable master branch with crazy changes;
- Commit your changes as often as possible, and always attach a clear description of change. When you don't remember what change you have made or cannot describe it in one sentence, you have done too much without committing;
- After committing, alway do a "push" so that your branch at the GitHub server is also updated. This saves lots of tears when your laptop is stolen or disk is wiped clean;
- Therefore: **branch - edit - commit - push**. If you collaborate with others on their repo, **fork - branch - edit - commit - push - send pull request**;

3 Preparation

- Download git, install it;
- There may be a graphic UI to config your username and user email, or you can do it via command line:

[1] http://readwrite.com/2013/09/30/understanding-github-a-journey-for-beginners-part-1#awesm= opzx0XktDVGJmx

```
$ git config --global user.name "Your␣Name␣Comes␣Here"
$ git config --global user.email you@yourdomain.example.com
```

4 Basic usage: set up local/remote repo w/o branching or forking

First, cd to the regular, local directory that you intend to turn into a git repository. Let's say the directory is called "toydir".

```
$ git init
```

This turns toydir into a git repository(repo for short). Only after this can you do other git operations for toydir. To know what happened exactly, run this command "ls -la", you can see a new ".git" sub-directory added to your directory. That is to say, if you remove .git by "rm -r", git will not recognize toydir as a git repository.

```
$ git help
```

For simple help or reminder; if it gets complex, do a google search;

```
$ git status
```

Check status of the repo. For example, untracked/uncached files that you can consider adding to track/cache(only tracked files will be part of the git repository by your commit action); recently tracked/cached files, etc.

```
$ git add $FILENAME
```

Git add is used both for new and newly modified files, and in both cases it takes a snapshot of the given files and stages that content in the index, ready for inclusion in the next commit. We then say the files are tracked/cached. Specifically, you need "git add" to:

- Notify git to track/cache new files. All files have to be tracked/cached before you can commit them to repository. Note that the physical presence of a file in the directory != presence of a file in git repository.
- Notify git of the content update of already tracked/cached files. Suppose "file1.txt" is tracked, if you edit the file, you still need to do "git add file1.txt", otherwise you won't have updated file1.txt in your repo. If you have just modified already tracked/cached files, you can use "git commit -a" which does "git add" and "git commit" in one step. In fact, I always do 'git commit -a -m "$UPDATE_MESSAGE" ', where $UPDATE_MESSAGE is message about the change, wrapped up in double quotes so that -m takes it as a whole. However, if you added new files to track/cache, you have to do both "git add" and "git commit".

Note that if you have a directory full of files and want to add them all to track/cache, you can just do "git add ." where dot represents everything recursively from the current directory, unless you have set .gitignore to ignore some file patterns;

```
$ git rm --cached $FILENAME
```

To remove file or a list of files from git cache so that it won't be in the git repo if you do the next commit. Again recall that local directory != git repository. Removing a file in directory != removing it from git cache: you need to also do "git rm" to completely get rid of it. Similarly, you can "git rm" while keeping the file around in the linux directory.

```
$ git commit -a -m "$UPDATE_MESSAGE"
```

After making changes to the directory (editing files, tracking&untracking files, etc), you need to commit the changes so that they are saved in your repository with a version number.

Always make sure that you have added new files to track/cache before you commit. Do "git add" very often, do "git commit" very often. This is even more important when you do branching or forking.

```
$ git remote add origin $GIT_OR_HTTPS_URL
```

You need to push your local repo to the "remote" at the GitHub server so that you have your updates on the cloud and no worries for computer failure. To do the push, you need to have the remote **beforehand** for the corresponding local repo. For instance, if I'd like to push toydir to GitHub server, I need to:

- Open browser, log into GitHub, click the "Repositories" tab and create a new repository, preferrably the same name as the local directory, therefore toydir.
- On linux terminal, cd to toydir, and do:

  ```
  $ git remote add origin https://github.com/yanggao1119/toydir.git
  ```

  This connects local repo toydir to the remote repo in GitHub server, which is by default called "origin".

```
$ git push $REMOTENAME $BRANCHNAME
```

Now, you can push to the remote at the GitHub server. You will be prompted with your GitHub username and password. I believe you can also set it once for all. Tell me how and I will update this tutorial.

You can do: "git push origin master", which pushes your master branch on the local repo toydir to the master branch of the origin remote repo on the GitHub server. Master has to do with branching, and origin has to do with forking, to be talked below.

If you just do "git push", git will push all branches to the origin repo(Yang: actually, I am not very sure). Since currently you have no branching or forking, "git push" is the same as "git push origin master".

5 Managing branches

Whether you are collaborating with others, it is always a good practice to have your stable version at the master branch, check out a temporary branch to work on so that if you mess up with the temporary branch, your master branch is safe; When you are done with the temporary branch, you can merge it back to your master branch.

Since you have not branched out, you only have the master branch:

```
$ git branch
* master
```

To add a branch called "experimental", and then run "git branch" to check:

```
$ git branch experimental
$ git branch
  experimental
* master
```

The * is placed at master, meaning that your current branch is still the master branch

To switch to the experimental branch, run:

```
$ git checkout experimental
```

Please check the "Managing branches" section in git-scm.com/docs/gittutorial . It has a clear explanation on how to work with branches.

Notes:

- During dev, you often need to checkout a new branch from master and switch to that branch. Here is a shortcut to do it in one command:

```
$ git checkout -b new_branch
```

- **When you are happy with the new branch, merge it back to master. Before "git merge", make sure the two branches to merge have both committed updates**.
- If two branches have changed the same part of the same file, there will be failure in automatic merging. Suppose you are at the master branch, you'd like to merge with the experimental branch but both branch edited the same line of "file1":

  ```
  $ git merge experimental
  Auto-merging file1
  CONFLICT (content): Merge conflict in file1
  Automatic merge failed; fix conflicts and then commit the result
  ```

  You need to resolve the conflict since this is beyond git's understanding. One way to do it is to open the file in conflict with your text editor and resolve the conflict manually. You will see conflict-marked area begins with <<<<<<< and ends with >>>>>>>. These are also known as the conflict markers. The two conflicting blocks themselves are divided by a =======. Just keep what you want, then commit to finish resolving the conflict. You don't need to do "git merge" again.
- "git merge" will only change the current branch, not the branch to be merged with. Therefore, if you are at the master branch, running "git merge experimental" will only update master.
- After merging, you may want to delete the new branch. To delete your local branch:

  ```
  $ git branch -D new_branch
  ```

  To delete remote branch:

  ```
  $ git push origin --delete new_branch
  ```

6 Using Git for collaboration: forking a repo

The GitHub Help tutorial is clear and short, see here: https://help.github.com/articles/fork-a-repo . Just follow it.

Here is an example. If longwencan forked yanggao1119's toydir repo and cloned it to her local directory, longwencan can cd to the local repo directory and confirm with:

```
$ git remote -v
origin  https://github.com/longwencan/toydir.git (fetch)
origin  https://github.com/longwencan/toydir.git (push)
```

Note that the origin points to her own repo, not yanggao1119's repo where longwencan forked from. To track yanggao1119's repo updates, longwencan can add a remote called "upstream" pointing to yanggao1119's repo:

```
$ git remote add upstream https://github.com/yanggao1119/toydir.git
```

After this change, longwencan has these remotes:

```
$ git remote -v
origin  https://github.com/longwencan/toydir.git (fetch)
origin  https://github.com/longwencan/toydir.git (push)
upstream    https://github.com/yanggao1119/toydir.git (fetch)
upstream    https://github.com/yanggao1119/toydir.git (push)
```

For longwencan to perform an update with the upstream in case yanggao1119 update repo:

```
$ git fetch upstream
From https://github.com/yanggao1119/toydir
 * [new branch]      experimental -> upstream/experimental
 * [new branch]      master     -> upstream/master
```

For longwencan to merge her master branch with the master branch of yanggao1119's repo:

```
$ git checkout master
# Switched to branch 'master'
$ git merge upstream/master
```

Merging between branches of different repos is similar to merging between branches within a repo, which we discussed in the previous section.

However, **it is better to checkout a new branch to work on**. So longwencan creates a new branch called "edit1" and switches to that branch.

```
$ git checkout -b edit1
```

After editing, adding and committing, longwencan can push to origin, which is her remote at the GitHub server:

```
$ git push origin edit1
```

After the push, longwencan wants yanggao1119 to update as well. Therefore longwencan logs in github website, select the "edit1" branch, and click at the green button for "Pull Request", and writes this message to yanggao1119: "I have updated file1 and added file7, blahblah".

When yanggao1119 logs into GitHub website, she can see 1 Pull Request on the right pane. It is yanggao1119's turn to merge the pull request.

To be safe, we'll always do merging via command line. So yanggao1119 selects "command line" in the pull request and follow the instruction commands, which is like this:

- Step 1: From your project repository, check out a new branch and test the changes.

  ```
  $ git checkout -b longwencan-edit1 master
  ```

  This creates a temporary branch called longwencan-edit1, and switches to it.

  ```
  $ git pull https://github.com/longwencan/toydir.git edit1
  ```

  This pulls updates from the edit1 branch in longwencan's repo toydir.git, which is exactly the branch that sends yanggao1119 the pull request. There may be conflicts, since longwencan-edit1 by creation is a copy of the yanggao1119's master branch. Resolve the conflicts manually as mentioned above, and then commit to finalize.

- Step 2: Merge the changes and update on GitHub.

  ```
  $ git checkout master
  $ git merge --no-ff longwencan-edit1
  ```

  Note that since we already resolved conflicts in longwencan-edit1, we can **just merge longwencan-edit1 back into master with –no-ff**. In this sense, longwencan-edit1 is a **bridge&buffer**. You can delete longwencan-edit1 after this merge.

  ```
  $ git push origin master
  ```

  This updates to yanggao1119's remote repo on GitHub server.

7 Using Git for close collaboration: multiple project admin

Markus Dreyer reminds me that forking&pull is too slow and complex for people that collaborate closely. Instead of forking repo, simple give "push and pull, and administrative access" to all members in the team.

Say, yanggao1119 and longwencan are in a team called "quickdev" in a project Muko2014. She would like to collaborate closely with longwencan on this repo: https://github.com/Muko2014/toy. So yanggao1119 just needs to go to https://github.com/Muko2014/toy/settings/collaboration by

clicking “Settings” then “Collaborators”, and make sure that the team ”quickdev” is added under ”Teams”.

After this change, longwencan can do all things as yanggao1119 can. longwencan needs to first clone the repo to local dir, with this command:

```
$ git clone https://github.com/Muko2014/toy.git
```

longwencan can commit and push directly to the origin/master just as yanggao1119 can. There is no need to send yanggao1119 a pull request.

If yanggao1119 updated the repo, longwencan can fetch the origin to the local dir by:

```
$ git fetch
```

or

```
$ git fetch origin
```

and then do a merge with origin/master by

```
$ git merge origin/master
```

don’t confuse with the “upstream” thing in the last section, which only makes sense in forking. Here the only remote repo is origin.

In real scenario, both yanggao1119 and longwencan might have updated the repo. So after “git fetch”, instead of doing a “git merge”, longwencan may wan t to do a “git rebase” to rebase her own commits on top of the updated origin/master which then automatically attempts merging, as if they happen in that chronological order. Here is the command:

```
$ git rebase origin/master
```

If this merge fails due to conflict, longwencan can resolve conflict in the file and then do

```
$ git add .
```

and

```
$ git rebase --continue
```

finally, longwencan needs to push to the remote repo by

```
$ git push
```

so that yanggao1119 can see the update.

Two things to remember for yourself and for your collaborator:

- Always do your code dev on a new branch[2] instead of your master branch. This translates to using “git checkout -b $FEATURE_NAME” very often. When you are happy about dev, checkout master, merge with your dev branch and then push;
- When doing dev on your new branch, do as many commits as possible, as I said before. However, if you simply do a merge and push, your collaborators would see too many edits that don’t make sense to them, such as adding or deleting a small variable. To be considerate, do a “squash” when you merge, and then commit with a more substantial message. That is to say:

  ```
  # done lots of edits on dev branch called new2, happy to merge
  $ git checkout master
  $ git merge --squash --no-commit new2
  $ git commit -m $SUBSTANTIAL_CHANGE_INFO
  $ git push
  ```

[2] you can hear some people referring to it as “feature”

8 Special issues

8.1 Wildcard

Git supports wildcard, i.e., the * char, which saves your time/effort. Suppose you'd like to add files "file1.txt" and "file2.txt" to track/cache, you can do:

```
$ git add file1.txt file2.txt
```

If you later find that you don't want to cache them, you can do:

```
$ git rm --cached file1.txt file2.txt
```

With wildcard you can simply do:

```
$ git add *.txt
$ git rm --cached *.txt
```

Note that **git wildcard is recursive by default, different from linux wildcard**. That is to say, "git add *" recursively add all files/directories into cache, and "git add '*.txt"' recursively add all .txt files into cache.

8.2 Protocol

Sometimes you will have problem with the protocol you are using. For example, when you use VPN and run these:

```
$ git remote add origin git@github.com:yanggao1119/toydir2.git
$ git push origin master
```

You may get this error: "Permission denied (publickey)". You can try to remove the old origin, and then use HTTPS to try push:

```
$ git remote rm origin
$ git remote -v
# to check if you have removed the old origin with the git protocol
$ git remote add origin https://github.com/yanggao1119/toydir2.git
$ git push origin master
```

I guess it is always better to use HTTPS, need to confirm.

9 Problems&suggetions

Contact Yang Gao via yanggao1119@gmail.com.

10 Ackowledgements

Markus Dreyer, Jon May

11 References

1. http://code.tutsplus.com/articles/team-collaboration-with-github–net-29876
2. http://readwrite.com/2013/09/30/understanding-github-a-journey-for-beginners-part-1#awesm= opzx0XktDVGJmx
3. https://help.github.com/articles/fork-a-repo
4. git-scm.com/docs/gittutorial