working directory, **staged snapshot**, and **commit history**

`git config`

```
cd /c/Users/"Yating Liu"/Desktop/Inf43Hw3
```

to change the current folder/directory

在windows和mac上，backslash(\)和forward slash(/)是不一样的，在文件名有空格时，使用引号(`""`)

```
git config --global user.name "Peter Anteater"
git config --global user.email "panteat@uci.edu"
```

To look at all your configuration information: `git config --global -l`

`--global`表示该电脑下的所有仓库都会使用这个配置

set up new repo locally

```
git init
```

create a local git repo

initialize empty Git repository in

```
git status (branch master)
git add file1.txt/git stage file1.txt
```

when you `add` a file you are telling Git to keep track of it.`add` also tells Git to **stage** the file, which means put it in the stage of being ready to be committed

`git stage`is really just another name for `git add`.

> `git add .` vs. `git add -A` vs. `git add -p`
>
> The difference is that `git add -A` also stages files in higher directories that still belong to the same git repository. `git add .` only affects the current directory and subdirectories.
>
> you use the `git add` command to stage a new or modified file. *However, to stage the deletion of a file, you need to use the **git rm*** So run the commands `git add file1.txt` and `git rm file2.txt` to set the stage
>
> And better than `git add .` is `git add -p` because it will interactively ask what to stage, this will show you all the changes again and will show you comment / logging that you forgot to remove as well. Also commit often, as large changes tend to be hard to review / oversee.

when you `commit`, you in effect copy all staged files to the repository. The `-m` is a flag (that's what the hyphen indicates) which tells Git that the following string is a message to record with the commit.

```
git commit -m "Committing a new file with my name"
```

the power of the `-a` flag is that it tells git to automatically stage all ***tracked***, ***modified*** files before the commit

```
git commit -a -m "Now has my major"/git commit -am "Now has my major"
```

amend your commit message

```
git commit --amend -m "Added favorite restaurant and movie"
```

`git reset`, `git revert` and `git checkout`

```
git reset --hard xxxx
git reset --hard HEAD       //(going back to HEAD)
git reset --hard HEAD^      //(going back to the commit before HEAD)
git reset --hard HEAD~2     //(going back two commits before HEAD)
git reset --hard HEAD@{1}   //undo a hard reset on Git
```

The purpose of the `git reset` command is to move the current HEAD to the commit specified (in this case, the HEAD itself, one commit before HEAD and so on).

the version after xxxx will be deleted, to push to remote repo, you have to use `git push -f` because remote origin still has HEAD points to the deleted commit

for more about `git reset`, check this stackflow thread with many details (`--hard`, `--mixed`, `--soft`)

```
git revert
```

in `git reset`, you delete the older commit and move the head backwards, in `git revert` you are introducing an order version to the current branch and move the head forward. When you merge with another branch, the changes may be introduced again in using reset, but will not be introduced if using revert.

This method would not have the disadvantage of `git reset`, it would point HEAD to newly created reverting commit and it is ok to directly push the changes to remote without using the `-f` option.

check this simple article to see the difference between `git reset` and `git reverse`

```
git checkout xxxx
```

replacing xxxx with the **first** four digits/characters from that hash (thankfully typing in the entire hash is not required). You will see a frightening message about a detached HEAD. git can keep track of separate, parallel, streams of edits to a project. Each stream of edits is called a branch, and a branch can have a name. For instance, multiple programmers who are working on and commiting changes to the same file will probably establish different branches. HEAD is git-ese for the current (not necessarily the last) commit in the current branch.**Since we've gone back in time and are potentially (but haven't yet) starting a new branch, HEAD is "detached" (from any established, named branch)**

This is useful for quickly inspecting an old version of your project. However, since there is no branch reference to the current HEAD, this puts you in a detached HEAD state. **This can be dangerous if you start adding new commits because there will be no way to get back to them after you switch to another branch.** For this reason, you should always create a new branch before adding commits to a detached HEAD.

```
git checkout HEAD~2 foo.py
```

the following command makes `foo.py` in the working directory match the one from the 2nd-to-last commit

Checking out a file is similar to using `git reset` with a file path, except it updates the *working directory* instead of the stage. Unlike the commit-level version of this command, this does not move the `HEAD` reference, which means that you won’t switch branches.

Just like the commit-level invocation of `git checkout`, this can be used to inspect old versions of a project—but the scope is limited to the specified file.

If you stage and commit the checked-out file, this has the effect of “reverting” to the old version of that file. Note that this removes *all* of the subsequent changes to the file, whereas the `git revert` command undoes only the changes introduced by the specified commit.

Like `git reset`, this is commonly used with `HEAD` as the commit reference. For instance, `git checkout HEAD foo.py` has the effect of discarding unstaged changes to `foo.py`. This is similar behavior to `git reset HEAD --hard`, but it operates only on the specified file.

> This is an update to the "Commit History" tree. The `git checkout` command can be used in a commit, or file level scope. A file level checkout will change the file's contents to those of the specific commit.
>
> A revert is an operation that takes a specified commit and creates a new commit which inverses the specified commit. git revert can only be run at a commit level scope and has no file level functionality.

> A reset is an operation that takes a specified commit and resets the "three trees" to match the state of the repository at that specified commit. A reset can be invoked in three different modes which correspond to the three trees.
>
> Checkout and reset are generally used for making local or private 'undos'. They modify the history of a repository that can cause conflicts when pushing to remote shared repositories. Revert is considered a safe operation for 'public undos' as it creates new history which can be shared remotely and doesn't overwrite history remote team members may be dependent on.

*check this to see the detailed usage of `git reset`, `git revese` and `git checkout` on commits and files. It's very detailed and complete.

```
git rm --cached filename.extension
```

参考这篇很清楚rm 、git rm 、git rm --cached的区别

It will not remove the file from the staging area only, but entirely from tracking. If you just want to remove a file from staging please use `git reset filename.extension`.

When invoked with a file path, `git reset` updates the *staged snapshot* to match the version from the specified commit. For example, this command will fetch the version of `foo.py` in the 2nd-to-last commit and stage it for the next commit:

```
git reset HEAD~2 foo.py
```

As with the commit-level version of `git reset`, this is more commonly used with HEAD rather than an arbitrary commit. Running `git reset HEAD foo.py` will unstage foo.py. The changes it contains will still be present in the working directory. The `--soft`, `--mixed`, and `--hard` flags do not have any effect on the file-level version of `git reset`, as the staged snapshot is *always* updated, and the working directory is *never* updated.

我个人觉得checkout只是单纯的把HEAD移走（但是因为HEAD和当前的branch detached了，可能会回不去，适合看order version），`git checkout file`是将file恢复到某个commit，并且工作区也会相应改变。reset是将最近的commit删除，并将HEAD往后移，reset的三种模式-hard，mix，soft分别对应工作区+stage snopshot+commit history，snopshot+commit hisotry(default)，commit history。`git reset file`是unstage某个file，reset file不会改变工作区的文件，但是checkout file会改变工作区的文件。revert是加上之前的commit并且将HEAD向前移达到撤回操作的效果。git rm会先删除工作区的文件，然后将删除的操作commit，而加上--cache则会在stage里删除，但是在working directory保留。每一个commit都对应一个log，每一个log都有HEAD做标记

`git log`

```
git log
```

this will display the history of changes made to the repository

```
git log -p -3
```

the `-p` flag will show you the diffs for each change. The -3 will limit what's displayed to the last 3 log entries

Note: A command such as `git log` sends text output to the shell using a bash command named *less* to display one windowfull of output at a time (*less* is named after a similar, earlier utility named *more*, in a classic example of hilarious techie humor). At the : prompt, you can press **h** for help, **q** to exit, **Enter** to advance one line, or **Space** to advance one screenfull

```
git log -p > git_log_partB.txt
```

the **>** is a shell command that redirects the output of the program on **>**'s left to the file named on **>**'s right.

```
git log -1 -p --before='2014-03-31 11:52:45'
```

this is nothing new except for the `--before='2014-03-31 11:52:45'` part. That tells git you only want to see log entries for changes made before **March 31, 2014 at 11:52:45am**. As you saw in part B, the `-1` means you only want to see one entry, and the `-p`means you want to see a diff of the changes

```
git log --reverse
```

since we have a complete copy of the Reddit project's repository, we also have a copy of every snapshot going all the way back to the beginning of the project. To see the log entries for the **earliest** commits

```
git log -1 4778b17e939e119417cc5ec25b82c4e9a65621b2
git show 4778b17e939e119417cc5ec25b82c4e9a65621b24
```

(Don't forget that you can use only the first four digits/characters of the hash. If git complains that the short SHA1 is ambiguous (because more than one commit has the same first four digits), try adding a few more digits from the long hash.)

```
git log --skip=100 -5
```

one more git log option to know about is --skip=N, where N is a non-negative integer. This means to skip N commits before starting to show the commit output

标签管理

用tag就能进行回滚

查看所有标签 `git tag`

创建标签 `git tag name`

指定提交信息 `git tag -a name -m "comment"`

删除标签 `git tag -d name`

标签发布 `git push origin name`

分支管理

create a new branch with name `git checkout -b branch_name`

see all the branch, the current branch is green with asterisk `git branch`

create a new branch `git branch branch_name`

switch to another branch `git checkout branch_name`

delete branch `git branch -d branch_name`

merge branch: `git merge branch_name`

Remote Git Repo set up

```
git remote
```

the list of remote repos connected to your local project

创建SSH key

```
ssh-keygen -t rsa -C "yatinl4@uci.edu"
cd /c/Users/liuyating
cd .ssh/
cd id_rsa.pub
ssh -T git@github.com
```

添加远程仓库

```
git remote add origin git@github.com:tylerdemo/demo4.git //url of the server
git pull origin master --allow-related-histories //master或其他branch
git push -u origin master //-u代表本地的master, master或其他branch
```

如果是clone，默认仓库是绑定上去的，所以`git clone`的repo，通过`git push`就可以了

```
git clone url
git clone https://github.com/reddit/reddit.git
```

You now have a local copy (on your computer) of the remote repository. It's important to understand that **this is not only a copy of the source code, but also a copy of the history of changes stored by git.**

`git remote add` vs. `git clone`

- **`git remote add` just creates an entry in your git config that specifies a name for a particular URL. You must have an existing git repo to use this.**
- **`git clone`creates a new git repository by copying an existing one located at the URI you specify.**
- `git clone` contains more commands

```
git fetch
```

compared to git pull, git fetch will not do merge

Other

(on github) pull request, and you can comment. If accepted, you can merge to the master branch and you can delete branch(that branch is gone only on the cloud)

```
touch .gitignore
```

create a file named **.gitignore**. You can use text editor to open this file, write down the files for directories that you want to ignore in repo. The ignored files will not be shown during execution.

So use this command when you don't want to upload some files.