Git/GitHub Tools for Research*

Wooyong Park

2025-05-09

Table of contents

*For the most recent version of this pdf, check out this website

1 Basic Features

1.1 Configuring Git

- `git config --list`: displays a list of customizable settings

Git has three levels of settings:

1. `--local`: settings for one specific project
2. `--global`: settings for all of the projects

3. `--system`: settings for every users on the computer

An example of `git config --list`:

```
(base) Tommysui-MacBookPro:~ tommy$ git config --list
credential.helper=osxkeychain
init.defaultbranch=main
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
user.name=wyeconomics
user.email=tommypark822@gmail.com
```

1.1.1 Changing the Settings

- `git config --global [setting] [value]`: changes the particular setting to the specified value

1.1.2 Creating repos

- `git init [reponame]`: creates a new repository
- `git init`: converts an existing project(directory) into a Git repository

1.1.3 Caution for nested repositories

Nested repo is a Git repo inside another Git repo. There will be two `.git` directories per project, so Git wouldn't be able to identify which to update.

1.1.4 Using an alias

By executing `alias.[aliasname]`, one can create an alias for committing files. This is typically used to shorten a command.

- `git config --global alias.[aliasname] [command]`: creates a global alias for the specified command

The following is a nice example.

```
git config --global alias.unstage 'reset HEAD'
```

Then entering `git unstage` would be equivalent to `git reset HEAD`.

1.1.5 Tracking aliases

Git tracks aliases by storing them in a `.gitconfig` file. One can access it by calling `git config --global --list`;

1.1.6 Ignoring specific files

We can ignore certain files by creating a file called `.gitignore`

- `nano .gitignore`: creates a `.gitignore` file.

Using the `.gitignore` file, we can specify which files should be ignored. For example, if we add `*.log` to the `.gitignore` file, Git will ignore any file ending with `.log`;

1.2 Erasing Git from a project

1.2.1 Removing Git

- `rm -rf .git`: removes Git from the project

1.2.2 Verifying Git is removed

- `git status`: verifies whether Git is removed

2 Version Control

- `pwd`: returns the current working directory
- `ls`: returns the files inside the current directory
- `cd [dirname]`: changes the directory to the specified directory name

2.1 Editing a File

- `nano [filename]`: can be used to delete, add, or change contents of a file
- `echo` : can be used to create/edit a file
  - **Create** : `echo [content] > [filename]`
  - **Edit** : `echo [content] >> [filename]`
- `Ctrl + O` & `Ctrl + X`: is the shortcut to save file

2.2 Checking Git Version

- `git --version`: returns which version of Git is installed

2.3 Repository (Git Project)

A Git project consists of two parts: (1) the files and directories we create and edit, (2) the Git storage with the name `.git`.

The combination of these two parts is called a repository, often referred to as a repo.

- **Staging**: Putting files in the staging area is like placing a letter in an envelope.
- **Committing**: Making a commit is like putting the envelope in a mailbox. After comit, you can't make any further changes.
- `ls -a`: shows the hidden files in the current directory, including `.git` file

2.3.1 Staging

- `git add [filename]`: adds a single file to the staging area
- `git add .`: adds all files in the current directory to the staging area

2.3.2 Committing

- `git commit -m "log message here"`: `git commit` makes a commit and the suffix `-m` adds *log message* for the commit.

2.3.3 Checking Status

- `git status`: tells us which files are in the staging area, and which files have changes that aren't in the staging area yet.

2.4 Comparing files

2.4.1 Unstaged Version vs Committed Version

- `git diff [filename]`:compares the unstaged version of a file to the last commit

The line with the two `@@` symbols tells us the location of the changes.
The line that starts with `-` symbol tells us the line that is erased in the unstaged version.
The line that starts with `+` symbol tells us what is added.

2.4.2 Staged Version vs Committed Version

- `git diff -r HEAD [filename]`: compares the staged version of a file to the last commit

`-r` indicates that we want to look at a particular revision of a file.
`HEAD` is a shortcut for the last commit of the file.

According to ChatGPT, the code above is wrong. ChatGPT recommends the following code instead.

- `git diff --staged`: compares the staged version to the last commit.

2.4.3 Multiple Staged Files vs Committed Version

- `git diff -r HEAD`: compares the staged version to the committed version of *all the files in the directory*

2.5 Storing data with Git

Git commits have three parts:

- **Commit**: contains the metadata
- **Tree**: tracks the names and locations in the repo
- **Blob**: binary large object(contains data of any kind, compressed snapshot of a file's contents)
- `git log`: displays all the commits made to the repo in chronological order, starting with the oldest.
- Press `space` to show more recent commits.
- Press `q` to quit the log and return to the terminal.

2.6 Git Hash

A hash is a unique indentifier that enables Git to share data efficiently between repos. If two files are the same, their hashs will be the same. Therefore, Git can tell what information needs to be saved in which location by comparing hashes.

To find a particular commit, we would open `git log`. After that, we copy the first 6-8 characters of the hash, and run `git show [hash 6-8 first characters]` to find out that particualr commit content.

- `git log`: opens the commitment log
- `git show [hash 6-8 first characters]`: shows the details of the commitment with the specified hash
- `git diff [hash1] [hash2]`: compares the two commitments with the specified hash

2.7 Viewing Changes

2.7.1 The `HEAD` shortcut

Use `~` to pick a specific commit to compare versions.

- `HEAD~1`: the second most recent commit
- `HEAD~2`: the third most recent commit

NOTE: must not use spaces before or after the tilde `~`

- `git show HEAD~3`: shows the details of fourth most recent commit
- `git diff HEAD~2 HEAD~1`: compares the third most recent commit and the second most recent commit

2.7.2 Changes per document by line

- `git annotate [filename]`: displays the detail of changes in commitment per document by line(hash, author, time, line#, line content)

2.8 Undoing changes

2.8.1 Unstaging a File

- `git reset HEAD [filename]`: unstages a single file from the staging area
- `git reset HEAD`: unstages all files from the staging area

Why do we need `HEAD` when unstaging files from the staging area? What we do through `git reset HEAD` is we instruct Git to match the staging area with the current commit state. Thus, we do need to call `HEAD` so that the last buffer zone(i.e. the staging area) matches the last commit.

2.8.2 Undoing changes to an unstaged file

- `git checkout -- [filename]`: reverses the unstaged file to the commited version
- `git checkout .`: reverses all unstaged files in current directory and any subdirectories to the commited version

`checkout` means switching to a different version(default to the last commit).

2.8.3 Reverting commits

Sometimes, we commit files that contains an error, and then spot the issue. Naturally, we need a command of restoring a repo to the state before the previous commit and make a new commit with the previous version. This is what `git revert` does. Also, the `git revert` command opens a text editor in the shell to add a commit message.

- `git revert HEAD`: reinstates the previous version and makes a commit(this restores all the files updated in the former commit).
- `git revert [hash]` or `git revert HEAD~[n]`: reinstates the version before `hash` or `n`th from latest commit and restore the previous version.

However, in other cases we might want to leave the revert in the staging area and don't commit them. `-n`, which stands for 'no commit' would do that.

- `git revert -n HEAD`: reverts the final commit without commiting

2.8.4 The sequence of unstaging, undoing changes, making changes, restaging, recommitting

```
git reset HEAD
git checkout .
nano [filename]
git add .
git commit -m "log message"
```

2.8.5 Restoring an old version of a file

- `git checkout -- [filename]`: reverts the unstored file to its last commit
- `git checkout [hash 6-8] -- [filename]`: reverts the unstored file to the specified commit
- `git checkout HEAD~1 -- [filename]`: reverts the unstored file to the second to last commit

2.8.6 Restoring a repo to a previous state

- `git checkout --`
- `git checkout [hash 6-8]`
- `git checkout HEAD~1`

2.8.7 Customizing the log output

If the project scale is large, `git log` alone would display excessive amount of commits. Therefore, customizing the log output would be adequate.

- `git log -3`: restricts the number of commits to three
- `git log -3 [filename]`: displays three most recent commit of the specified file
- `git log --since='Month Day Year'`: displays only the commits made since the specified date
- `git log --since 'M D Y' --until='M D Y'`: displays only the commits made between the specified dates

2.8.8 Cleaning a repository

- `git clean -n`: displays which files are not being tracked
- `git clean -f`: deletes the files that are not being tracked

3 Working with Branches

Git uses **branches** to systematically track multiple versions of files. Branches are how directories are segmented in the process of their commit.

When working on projects, developing across different components is common. For example, while one part of the team can handle errors and bugs, other parts could try out new features. This is the reason having multiple branches is helpful, as it allows us to keep making progress concurrently. Suppose we want to test some new ideas, but we don't want to change our existing code until we have confirmed it works. If a certain work of a branch is done, then it could be merged to the branch that is provided to users, which is done by merging one branch to the `main` branch.

One can think of the `main` branch as the **ground truth**. Each branch exists for a specific task, and once the task is complete and the process is confirmed to be ground true, it is then merged to the `main` branch.

- `git branch`: displays what branches exist for the project(one with the asterisk`*` is the branch the user is currently at)

- `git branch [branch]`: creates a new branch
- `git branch -m [old name] [new name]`: renames the branch
- `git switch [branch]`: switches the branch we work in
- `git switch -c [branch]`: creates a new branch and moves to it
- `git branch -d [branch]`: deletes the branch
- `git diff [branch1] [branch2]`: displays the differences between branches
- `git checkout [branch]`: moves us to the specified *existing* branch

3.1 Merging branches

Suppose we want to merge two commits from different branches. Then,

- `source`: is the branch we want to merge **from**
- `destination`: is the branch we want to merge **to**
- `git merge [source] [destination]`: merges the source commit to the destination commit

3.2 Handling Conflict

3.2.1 Git conflicts

If the same file is both edited in two different branches, merging them will cause an error. After merging, we can locate the source of conflict by opening the file with `nano [filename]`.

```
<<<<<<< HEAD
=======
A) Write report.
B) Submit report.
>>>>>>> [branch2]
C) Submit expenses.
```

- `<<<<<<< HEAD`: indicates that the lines beneath it contain the file's contents in the latest commit of the *current branch*.
- `=======`: refers to the center of the conflict. This being right below the first line indicates that the lines beneath it are part of the file versions in the latest commit of the current branch. However, if the equal signs are after some content, this means that the two files have different content on the same lines in different branches.
- `>>>>>>>`: indicates the second branch.

3.2.2 How do we avoid conflicts?

- Prevention is better than cure.
- Use each branch for a specific task.
- Avoid editing a file in multiple branches.

4 Collaborating with Git

4.1 Pulling remotes

- **Local Repos**: are repos stored on the computer
- **Remote Repos**: help us collaborate with colleagues(mostly through **Github**)

4.1.1 Cloning a repo

Cloning is copying existing repos, local or remote, to the local computer's current working directory.

- `git clone [local repo path]`: clones a local repo
- `git clone [local repo path] [new name]`: clones a local repo and gives new name
- `git clone [URL]`: clones a remote repo
- `git clone [local repo path] [new name]`: clones a local repo and gives new name
- `git clone [URL]`: clones a remote repo
- `git clone [URL] [new name]`: clones a remote repo and gives new name

Whenever we clone a repo, Git stores a remote tag in the new repo's configuration to track where the original was. If we are in a repo, we can check the names of its remotes through `git remote` or `git remote -v`.

4.1.2 Naming a remote

When cloning, Git automatically names the remote `origin`.
We can manually name the remote like writing `\label{tag}` in LateX to create labels.

- `git remote add [name] [URL]`: names the remote repo

4.1.3 Gathering from a remote

`git fetch` fetches all branches from the remote into the local repo. A branch only in the remote will be newly created to the local repo.

- `git fetch [remote name]`: fetches a Git remote with the specified name into the current local repo
- `git fetch [remote name] [branch name]`: fetches a Git remote's branch with the specified remote and branch name
- `git merge [remote name] [branch name]`: merges the local repo to the remote's branch with the specified remote and branch name

`git pull` is equivalent to requesting both `git fetch` and `git merge`.

- `git pull [remote name] [bvranch name]`: fetches the Git remote and merges the local repo to the remote's branch with the specified remote and branch name

It is important to save locally before pulling from a remote. If there is a change made in the local repo that is neither staged nor committed and then try to pull a remote, Git tells us that local changes would be overwritten and aborts the pull.

REMARK) Cloning and fetching works for the *entire* local repo. However, merging and pulling the remote resources happen only to the destination local *branch*.

4.2 Pushing to a remote

4.2.1 Pushing to a remote

After saving changes locally, one can push the new repo to a remote. * `git push [remote name] [local branch]`: pushes the updated local to the specified remote

4.2.2 Push/pull workflow

Pulling a remote into the local repo comes before pushing the local to a remote. If you don't start the workflow by pulling from the remote, Git can display conflicts. Thus, you have to start with pulling a remote first.