GIT INSTRUCTION MANUAL

EVAN BERGERON
SUNNY GAKHAR
NISHAD GOTHOSKAR
FREDERICK LEE
ZIYANG WANG

Contents

Introduction

This is a manual on how to set up Git on your computer and set up a basic Git workflow. The document covers installing and setting up Git and how to work with Git. The majority of this document is primarily intended for users who want to learn Git to use it in a fast-paced setting such as a hackathon, this document is also useful as a reference for experienced Git users who want to refer to some specifc concept or command which they need.

What is Git

Git is a version control system designed to be used for working on small and large projects. It can be used for tracking changes between files and coordinating work on project files among multiple people.

Motivation

When you were in school and you had short homework assignments, you would just start them and finish them in a short span of time (not longer than a week). But when you move on to designing and working on bigger projects with other people, there are multiple issues that come into play. Say you are participating in a hackathon and have finalized your idea and distribution of work among the teammates. How do you actually work on the project together?

Having all of them work on one computer is not optimal. You might have each teammate work on his own piece independently, but how do you merge everyone's work? Moreover, what if two or more teammates work on the same file, but do different modifications unknown to the others? And what if someone wants to explore a different direction to work on, while keeping the original work intact? Enter Git.

Benefits

(1) Using Git, if you're working on a project, you can "commit" the changes you have made and Git will keep track of all your commits.
(2) If you want to explore a new direction of work which you don't want to integrate with your main project just yet, you can create a new branch and work on the branch without disturbing your main project. You can easily switch between multiple branches to work on multiple features, can when the time comes, you can merge with the main branch.

Installing Git

Windows

If you have Windows, one easy way of installing Git is from this website:

`https://git-scm.com/download/win`

Once you have downloaded the installation file, you can run it and proceed through the installation steps.

Mac

Similar to Windows, one way of installing Git on is from this link

`https://git-scm.com/download/mac`

Another way of installing Git is using the Xcode Command Line Tools. Open up Terminal and simply type `git`. If you dont have Git installed already, it will prompt you to install it.

Linux

If you're working on Linux, you can install Git using a basic package management tool that comes with your distribution.

Debian/Ubuntu
```
$ apt-get install git
```

Fedora
`$ yum install git` (up to Fedora 21)
`$ dnf install git` (Fedora 22 and later)

Arch Linux
```
$ pacman -S git
```

FreeBSD
```
$ pkg install git
```

Solaris 9/10/11 (OpenCSW)
```
$ pkgutil -i git
```

Solaris 11 Express
```
$ pkg install developer/versioning/git
```

OpenBSD
```
$ pkg_add git
```

Initializing Repository

To make sure you have Git set up, type `git` into your console (Terminal for Mac/Linux users and Command Prompt for Windows users) and the following should show up (the full output has been elided here).

```
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
[--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
[-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
[--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
<command> [<args>]

...

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

To start a new repository, first go to ttype `git init` into the console. It should output the following:

```
$ git init
Initialized empty Git repository in <path to current directory>/.git/
```

This creates a `./git` directory in your current directory, which consists of all information about the repository. It consists of a HEAD file, which points to the current version of the repository.

Adding files to version control

Committing changes

Reverting changes

Branching

In git, branching is used to keep track of the different paths of development for your project. The master branch is the main branch of your project and after features or changes are verified, they are usually added to the master branch of the project.

To create a new branch (for example if you want a new branch named addfeature):

```
$ git branch add_feature
```

This creates a new branch in the development tree. Now you can work on this branch of the project without effecting the master branch.

As the figure above shows, branching creates a separate flow. Now in order to bring your current local state to that branch you will have to check it out by:

```
$ git checkout add_feature
```

Now the changes you make will be modifying this new branch rather than the master branch. To go back to the master branch you will use the same checkout command with master rather than addfeature.

Note that to checkout a different branch your current changes must be committed or stored away before you can switch.

A simple shortcut is to run:

```
$ git checkout -b add_feature
```

Note that to checkout a different branch your current changes must be committed or stored away before you can switch.

Now when you are done with development you can delete your branch by using:

```
$ git branch -d add_feature
```

If you want to rollback and see the state of your project at some previous point, you can use git log to look at the commit history and find the commit hash

and use:

```
$ git checkout 1c0191a6a6264f2e90f6905636321fde897b86df
```

To checkout the previous commit and see the state of the project at that point.

Merging Branches

Now after working on these separate branch and adding your new feature, you want to merge this work into your master branch so it will persist and new branches can build upon or use the feature you added. This is what merging does. It takes a branch and merges it into another branch. This can be done by first checking out the branch you want to merge INTO:

```
$ git checkout master
```

And then merging the other branch in by:

```
$ git merge add_feature
```

This may happen immediately, unless there are conflicts between the two branches that need to be resolved before they can be successfully merged. By using git status, you can see in which files these conflicts are located. Then within the files you will find sections of the form:

```
<<<<< HEAD
stuff in the current branch
=====
things in other branch
>>>>> add_feature
```

And you will have to replace these sections with the content you want and then commit the changes to fully process the merge of the two branches.

Ignoring Files

The best way to ignore files in a repository is to create a `.gitignore` file at the base of your repository with the names of all files to ignore.

Often you will find that in your projects, you will have files in your repository that do not need to be tracked. These include automatically generated files, larger libraries that are really external dependencies, or system specific files such as the `.DS_Store` file in Macs.

For example, let us have a repository with two files: `code.py` and `generated.txt`. Here we only want to keep track of changes to `code.py` while ignoring changes to `generated.txt` which is automatically generated every time `code.py` is run.

Without a `.gitignore` file, running `git status` would get the following output:

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    code.py
    generated.txt

nothing added to commit but untracked files present
(use "git add" to track)
```

We create a `.gitignore` file with the following content:

```
generated.txt
.gitignore
```

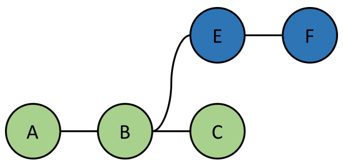
Running `git status` again, we see that the generated file and the `.gitignore` itself are ignored by Git as desired.

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    code.py

nothing added to commit but untracked files present
(use "git add" to track)
```

Rebasing

The rebasing and merging are the two primary ways of combining changes from separate branches with Git. One primary advantage of rebasing is that the Git history is made more linear which makes it easier to track changes over time.

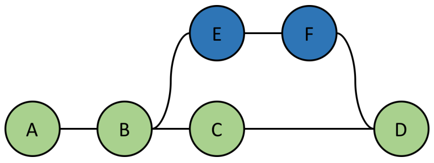
For example, say that you have been working on a separate branch called `feature` (shown in blue) on which you have two commits. At the same time, a friend of yours has committed change “C” to master (shown in green).



A regular merge command such as

```
$ git checkout master
$ git merge feature
```

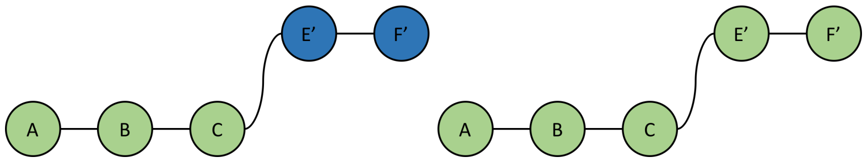
would result in the following Git history:



However, if you want a more linear history, you would ideally want your commits “E” and “F” to be stacked on top of your friends commits. To do as such, simply run as follows:

```
$ git checkout feature
$ git rebase master
$ git checkout master
$ git merge feature
```

The initial rebase command changes the root of your `feature` branch to your friend’s last commit as shown below on the left image.

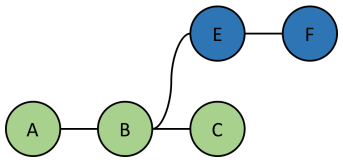


The final merge of your feature onto master then forwards master’s history to match your feature branch as shown above on the right. As you can see, the history follows a much more linear pattern compared to the branching pattern of a regular merge alone.

Cherry-picking

Cherry-picking is particularly useful when you don't want to apply all changes from a branch to another branch through a merge or rebase, but only want to apply a few specific commits.

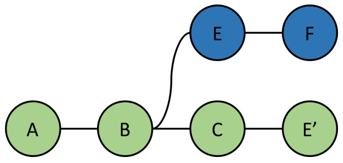
For example, take the hypothetical situation where your most recent commit "F" in your feature branch (shown in blue) is suffering from a application-breaking bug, but your manager wants your changes in commit "E" to be rolled out by this afternoon.



To apply commit "E" by itself to the master branch (in green), first find the commit-hash of your desired commit by running `git log` and searching for your commit. A commit-hash is an identifying string of letters and numbers that may look something like `f3def414605008d5c899b2691ff1b6d1d3798a0a`. After you have found your commit-hash, to apply your desired commit to the master branch, run the following commands:

```
$ git checkout master
$ git cherry-pick <commit-hash of E>
```

Running these commands gets your final desired result shown below:



where commit "E" has been applied to the master branch from your feature branch without merging your entire bug-ridden feature branch to master.

Collaboration with Github

You can use github for a more user-friendly version controlling experience. Almost everything you can do using the command line can be done using the desktop client. You can easily select files, read their diffs, and commit them.

Using the online interface, you can create repositories, add contributors, and track progress very easily.

The workflow using github usually involves many collaborators who make pull requests to the managers of the repository.

General workflows

Easily Reordering Commits

We've all been there. You pull from master, take several minutes to clean up the various merge conflicts, and then are ready to push. You pull one last time before pushing, and what do you know - someone has pushed again in the last couple minutes.

Previously, my workflow for this situation would look something like

```
$ git checkout -b tmp
$ git checkout master
$ git reset --hard HEAD˜
$ git cherry-pick tmp
```

This works fine, but theres a much easier way – one that involves very little typing. We can simply say

```
git rebase -i HEAD 2
```

This will open an interactive git rebasing session (the `-i` stands for interactive). The window will display something along the lines of

```
pick 370e221 Commit one
pick c342396 Commit two
```

In whichever text editor were in, we may simply reorder these lines to reorder the commits. Much shorter!

Adding partial files

I just used this, actually. Suppose youve changed a single file `foo.c` in different sections, and each of these changes are logically different. For instance, maybe you refactor some function `foo`, while at the same time fixing a bug in function `bar`. Rather than create a separate branch and manually edit the files, we can simply say

```
git add -p foo.c
```

This will bring up an interactive prompt. It will automatically cycle through all the different areas of the diff, asking you if you want to stage each section. You may hit `y` or `n` for yes or no.

Once youre done adding the subset of changes you want to commit, you can double-check you have the right changes staged by saying

```
git diff --cached
```

Once youre sure that youre good to go, just commit your changes as normal. You can repeat this process for the remaining changes. (Or just do a normal `git add` at this point).

Git aliases

Git aliases are a good way to save yourself a lot of typing. I frequently want to see the git commit history, but dont especially care about the body of each commit. Heres the command Ive added to my configuration:

```
git config --global alias.l "log --oneline"
```

I can then just type `git l` to see a one line log of this commit history. If there are several long commands you use frequently, this can be a great way to save yourself some time.

Vim workflows

At the time of writing, perhaps the most feature complete vim-git plugin is Tim Popes vim-fugitive. Consequently, we will assume usage of this plugin throughout the entire vim workflows tutorial.

Installing vim-fugitive

There are a number of ways to install vim-fugitive. The one suggested by Tim Pope is as follows:

```
$ cd ~/.vim/bundle
$ git clone git://github.com/tpope/vim-fugitive.git
$ vim -u NONE -c "helptags vim-fugitive/doc" -c q
```

Vundle is a great plugin manager for vim – if you use this, you may simply add the line

```
Plugin ’tpope/vim-fugitive
```

to your vimrc and run the `PluginInstall` command.

Easy git blame

Youre browsing some file and discover a horrible bug written by one of your coworkers. Youre about to storm over to someones desk and verbally abuse them for producing incorrect code. Before you deliver your diatribe, you need to know who to blame.

Before using `vim-fugitive`, you would have to exit vim, manually type `git blame <filename>`, and then search for the relevant line in the output. Now, you can simply type `:Gblame` in your vim prompt, and a vertical split will open up right next to the line in question. You could even establish a keybinding to do this for you! What was once several lines of typing is now a single keystroke away! Your coworkers have never been so scared...

Emacs workflows

Similarly to vim-fugitive for vim, Magit is (at the time of writing), the most feature-complete git wrapper for emacs. We will thus assume usage of this package.

Installing Magit

Perhaps the easiest way to install is through MELPA. Just run

```
M-x package-install RET magit RET
```

Using Magit