



Introduction Tutorial to Git

QI Group Meeting

Yoann Piétri

10/02/2023

LIP6 - Sorbonne Université - CNRS

What is Git?

Git is a free and open source **distributed version control system**.

It is also impossible to change any file, date, commit message, or any other data in a Git repository without changing the IDs of everything after it.

The Pro Git book

- Snapshots, Not Differences
- Nearly Every Operation Is Local (*Sorry non-locality*)
- Git Has Integrity (*Not like us*)
- Git Generally Only Adds Data
- The Three States (Working directory, Staging area, Repository)
- Git is cool

What is not Git?

Git is not:

- A magic solution to everything;
- A backup system¹;
- A synchronisation system;
- Github, Gitlab or any other tool built on top of Git;

¹In some specific cases, it may be used at such

Other version control tools

Git is not alone in the world of version controls and here are examples of some other tools:

- Subversion (svn)
- CVS
- BitKeeper
- Fossil
- RCS
- ...

Git remains the most use version control system today.

Good reads on Git:

- The Pro Git book <https://git-scm.com/book/en/v2> (many things of the presentation are inspired from there)
- The git reference manual <https://git-scm.com/docs>

Git basics

Start of a project example

```
1 $ git init -b main
2 $ git config --local user.email "Yoann.Pietri@lip6.fr"
3 $ touch README.md
4 $ git add README.md
5 $ git commit -m "Initial commit"
```

- | | |
|---|--|
| 1. Initialize the repository with branch main | 4. Add the file to the staging area (see later), i.e. what's going to be committed |
| 2. Configure the local (for current git repository) email | |
| 3. Create a file name README.md | 5. Commit |

What is a commit?

A commit is a **snapshot** (i.e. a picture) of your code at a given moment.

Unlike other version control systems, Git does not only store the difference between versions, each commit represents the full code, and git think you repository as a series of commits.

It makes git a miniature file system, with version control, integrity and powerful tools.

Snapshots, Not Differences

Let's show the commit history of our simple earlier project:

```
1  $ git log
2  commit 13d588fd13f50a8554dfc6cb06d83b689c82ef81 (HEAD -> main)
3  Author: Yoann Piétri <Yoann.Pietri@lip6.fr>
4  Date:   Wed Dec 28 11:08:29 2022 +0100
5
6  Initial commit
```

What is 13d588fd13f50a8554dfc6cb06d83b689c82ef81 ?

At commit, Git takes into his snapshot everything in the staging area, along with some metadata (commit name, committer information, date etc) and secure it in a secure manner and **generates a unique checksum**.

Any change on the commit will make this checksum change.

Let's show the commit history of our simple earlier project:

```
1 $ git commit --amend -m "Almost intial commit"
2 $ git log
3 commit 95129c0bd7ed42a458773d9fa4d6406d268a6161 (HEAD -> main)
4 Author: Yoann Piétri <Yoann.Pietri@lip6.fr>
5 Date: Wed Dec 28 11:08:29 2022 +0100
6
7 Almost intial commit
```

95129c0bd7ed42a458773d9fa4d6406d268a6161 \neq
13d588fd13f50a8554dfc6cb06d83b689c82ef81

It is also impossible to change any file, date, commit message, or any other data in a Git repository without changing the IDs² of everything after it.

The Pro Git book

Git Has Integrity

²The checksums are also the IDs of the git objects

Not screwing things up

When you do actions in Git, nearly all of them only add data to the Git database. It is hard³ to get the system to do anything that is not undoable or to make it erase data in any way.

This makes using Git a joy because we know we can experiment without the danger of severely screwing things up⁴.

The Pro Git book

Git Generally Only Adds Data

³Hard doesn't mean impossible

⁴Still possible though

The (almost) 3 states

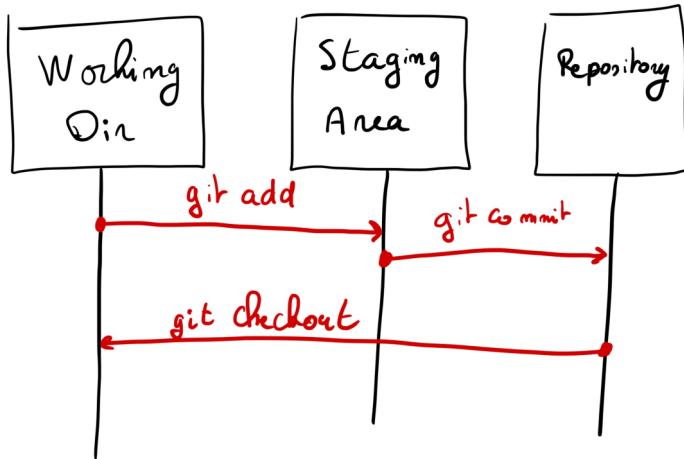
The 3 states

Git has 3 states and 3 associated regions

- Modified
- Staged
- Committed
- Working Directory
- Staging area
- Repository

The 3 states

The 3 states



The hidden state

What if, in the middle of implementing a feature, you need to work on another branch, or implement a hotfix ? Stashing is here for you.

Stashing takes the dirty state of your working directory and saves it on a stack of unfinished changes that you can reapply at any time (even on a different branch).

The Pro Git book

The hidden state



The hidden state

```
1  $ git status
2      On branch main
3      nothing to commit, working tree clean
4  $ touch unfinished_work
5  $ git status
6      On branch main
7      Untracked files:
8          unfinished_work
9
10 $ ls
11     README.md  unfinished_work
```

The hidden state

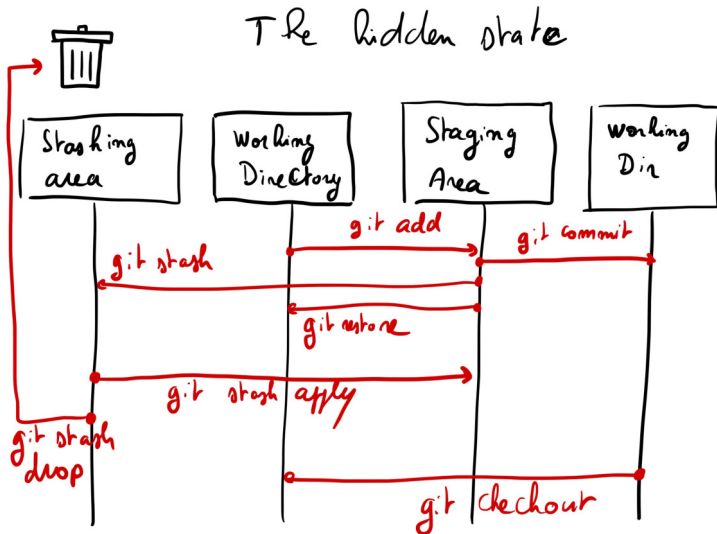
```
1  $ git add unfinished_work
2  $ git stash
3      Saved working directory and index state WIP on main: b0f26d4 Initial commit
4  $ git status
5      On branch main
6      nothing to commit, working tree clean
7  $ ls
8      README.md
9  $ git stash list
10  stash@{0}: WIP on main: b0f26d4 Initial commit
```

The hidden state

```
1  $ git stash apply
2      On branch main
3      Changes to be committed:
4          new file:   unfinished_work
5  $ ls
6      README.md  unfinished_work
```

```
1  $ git stash drop
2      Dropped refs/stash@{0} (3fd9e24718f373740c72398833fb1be353586246)
```

The hidden state



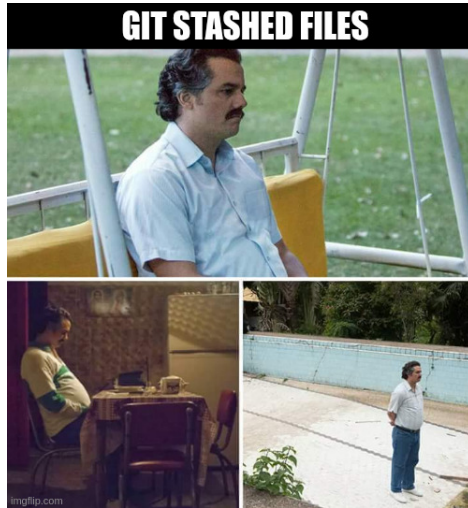
The Three States

Pay attention now - here is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states [...]

The Pro Git book

The Three States

The forgotten modifications




Branches


What is a commit?
(déjà-vu ?)

A commit is a **snapshot** of your staged work along with some metadata for the commit (author, data, message) and **one or more parent commit(s)**.



Add file1


Browse SourceActions

 main

 **Yoann Piétri** 2 months ago

parent [74d53cdeac](#) commit [6c38de9a1b](#)

 Signed by:  nanoy

 GPG Key ID: 6F8C39DA3C16888E

Parents



Branch

```
1  $ git checkout -b main2
2      Switched to a new branch 'main2'
3  $ touch file2
4  $ git add file2
5  $ git commit -m "Add file2"
6  $ git checkout main
7      Switched to branch 'main'
8  $ touch file1
9  $ git add file1
10 $ git commit -m "Add file1"
```

```
1 $ git log --graph --oneline --all
2     * 6c38de9 (HEAD -> main) Add file1
3     | * ed86875 (main2) Add file2
4     |/
5     * 95129c0 Almost intial commit
```

What is a branch?

A branch is a **lightweight movable pointer** to one of the project's commits.

When you commit on a branch, the pointer moves to the new commit.

When you create a commit from a branch A, you create an additional pointer B that points to the same commit as A, at the beginning.

What is HEAD?

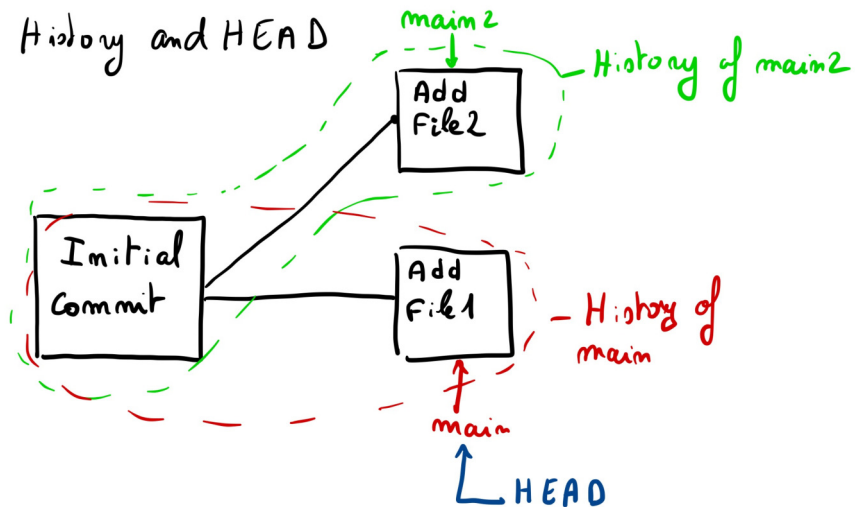
HEAD

Head is special pointer pointing to the current branch (could also point to a specific commit in certain case). It's also where the next commit will be added.

```
1 $ git log --graph --oneline --all
2     * 6c38de9 (HEAD -> main) Add file1
3     | * ed86875 (main2) Add file2
4     | /
5     * 95129c0 Almost intial commit
```

```
1 $ git checkout main2
2 $ git log --graph --oneline --all
3     * 6c38de9 (main) Add file1
4     | * ed86875 (HEAD -> main2) Add file2
5     | /
6     * 95129c0 Almost intial commit
```

HEAD



**Bringing back those branches
together**

Why branches?

Branching means you diverge from the main line of development and continue to do work without messing with that main line.

The Pro Git book

At some point, you might want to integrate your change in the main line of development.

Merging

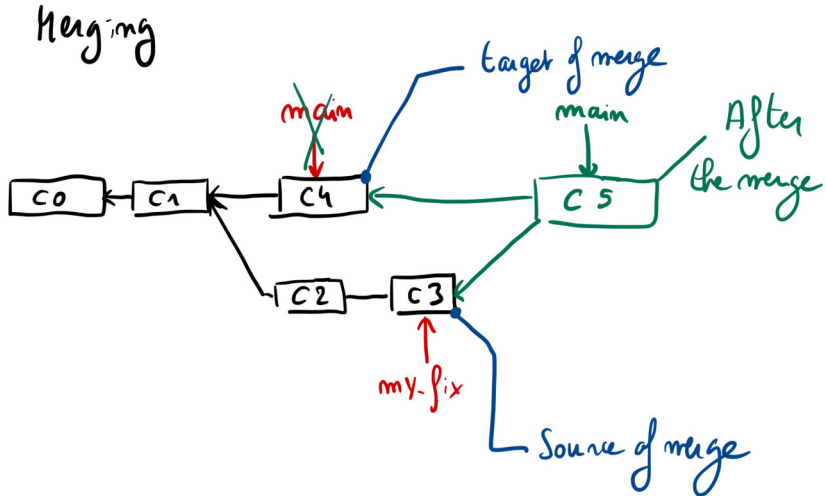
- is asymmetrical (one target, and one source)
- creates a merging commit (having the target and source commit as parents)
- is not destructive (the source branch still exists afterwards)

Merge

```
1 $ git log --graph --oneline --all
2     * 6c38de9 (HEAD -> main) Add file1
3     | * ed86875 (main2) Add file2
4     | /
5     * 95129c0 Almost intial commit
```

```
1 $ git merge main2 -m "Merge commit"
2 $ git log --graph --oneline --all
3     *   39bd944 (HEAD -> main) Merge commit
4     | \
5     | * ed86875 (main2) Add file2
6     * | 6c38de9 Add file1
7     | /
8     * 95129c0 Almost intial commit
```

HEAD





Merge conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging, Git won't be able to merge them cleanly.

The Pro Git book

You have to manually select which changes you want to keep from each branch and then do the merge commit.

In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase.

The Pro Git book

Merging

- Non-linear history
- Keep all previous commits
- Create a merge commit

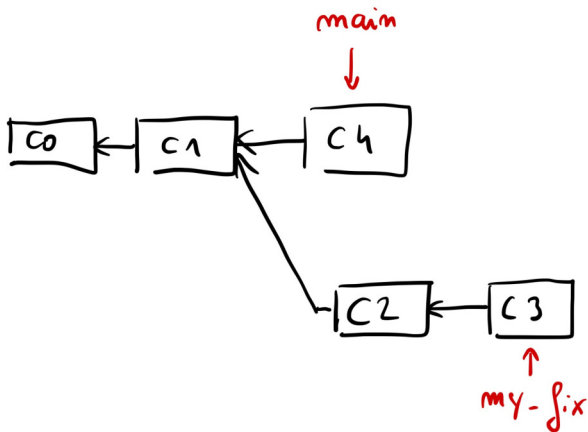
Rebasing

- Linear history
- Create new commits with same modification and delete old commits
- Does not create a merge commit

Before rebase

```
1  $ git log --graph --oneline --all
2      * 3fb7dfb (HEAD -> main) C4
3      | * 6008b93 (my_fix) C3
4      | * e60ea6c C2
5      | /
6      * adb048b C1
7      * ccee678 C0
```

Releasing 1

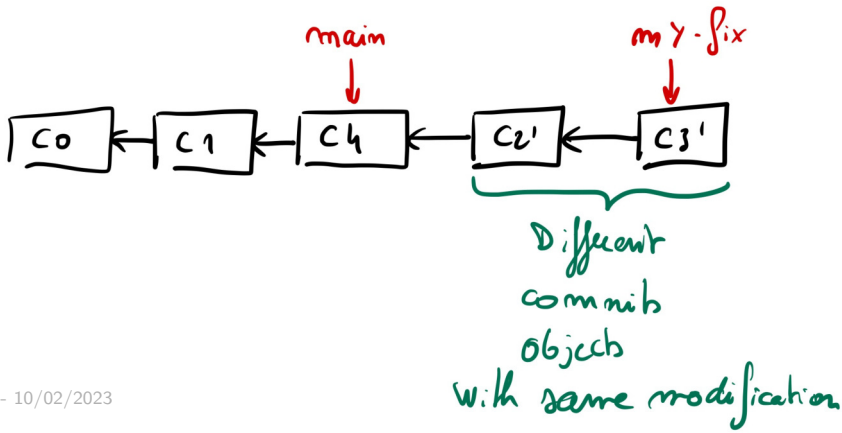


Rebase my_fix on main

```
1 $ git checkout my_fix
2 $ git rebase main
3 $ git log --graph --oneline --all
4     * 6efb6ee (my_fix) C3
5     * 7bece1a C2
6     * 3fb7dfb (HEAD -> main) C4
7     * adb048b C1
8     * ccee678 C0
```

Linear history, but checksums different.

Après rebase de my-fix sur main
Rebasing 2

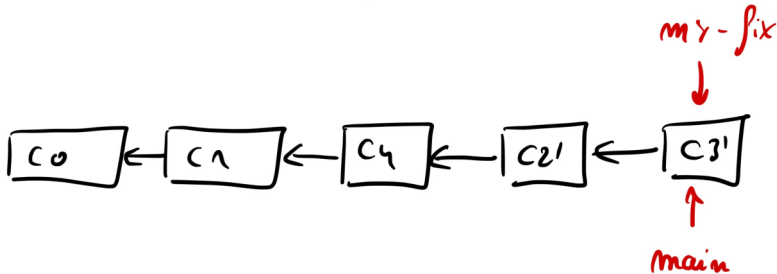


Fast-forward merge

```
1 $ git checkout main
2 $ git merge my_fix
3     Updating 3fb7dfb..6efb6ee
4     Fast-forward
5 $ git log --graph --oneline --all
6     * 6efb6ee (HEAD -> main, my_fix) C3
7     * 7bece1a C2
8     * 3fb7dfb C4
9     * adb048b C1
10    * ccee678 C0
```

Fast-forward: merge that does not create a merge commit (just moves the pointer, in case of a linear history).

Rebasing 3
Fast forward merge de my-fix sur main



Fast forward doesn't create a merge
Commit.

What if someone started a branch on a commit that was rebased after ?

The Perils of Rebasing

Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

Do not rebase commits that exist outside your repository and that people may have based work on.

If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family.

The Pro Git book

Revert and blame

What if someone (including you) made a bad commit ? You can easily correct by reverting a commit.

Revert will create a commit for you that undo the modification.

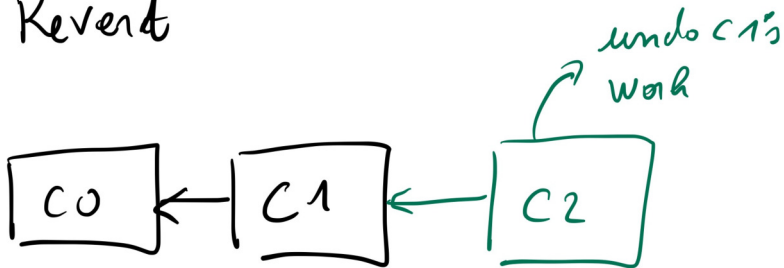
You should (almost) never just delete a commit and force push it⁵

⁵You can delete commits as long as they have not been pushed to remotes.

Revert

```
1      $ touch C1
2      $ git add C1
3      $ git commit -m "C1"
4      $ ls
5          C0  C1
6      $ git log --graph --oneline --all
7          * 76cb4f7 (HEAD -> main) C1
8          * 0f3e78a C0
9      $ git revert 76cb4f7
10     $ ls
11         C0
12     $ git log --graph --oneline --all
13         * fb72e86 (HEAD -> main) Revert "C1"
14         * 76cb4f7 C1
15         * 0f3e78a C0
```

Revert



Who fucked up this code?

git blame shows what revision and author last modified each line of a file

```
1 $ git blame alice.py -L 61,63
2     f7fafba9 (Yoann Piétri 2023-02-05 14:43:50 +0100 61)
3     f7fafba9 (Yoann Piétri 2023-02-05 14:43:50 +0100 62)
4     f7fafba9 (Yoann Piétri 2023-02-05 14:43:50 +0100 63)
```

Blame



Local and non-local operations

Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you.

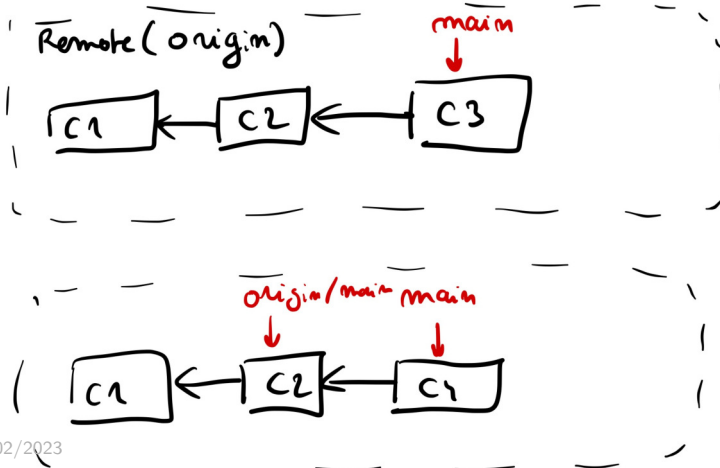
The Pro Git book

We assume the name of the remote to be origin.

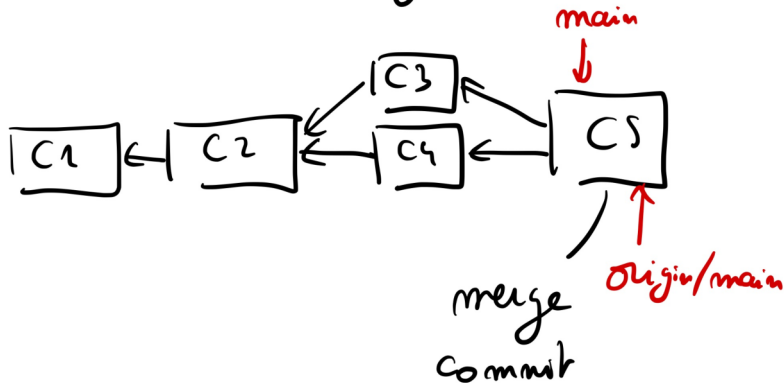
- `git remote show origin`: inspect the remote origin
- `git push`: Push your work to the remote
- `git fetch`: Fetch the new work form the remote
- `git pull`: Fetch the new work and merge it to the local branch

If there is no new commit, `git pull` will only do a fast-forward merge and no merge commit will be created.

Pulling with diverging histories



After pull (with merge behavior)



**As a rule of thumb, never force push to a remote repository
(I see you Paolo)**



Git is local

Most operations in Git need only local files and resources to operate - generally no information is needed from another computer on your network.

To browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you - it simply reads it directly from your local database.

This also means that there is very little you can't do if you're offline or off VPN. If you get on an airplane or a train and want to do a little work, you can commit happily.

The Pro Git book

Nearly Every Operation Is Local

Conclusion

Some cool stuff we didn't talk about

- Reset
- Tags
- Hooks
- Git on the Server
- Signing
- Cherry-picking
- Submodules
- Internals of Git
- Rewriting history (other than rebase)
- Stuff I don't even now about

Conclusion

Git is a free and open source **distributed version control system**.

It is also impossible to change any file, date, commit message, or any other data in a Git repository without changing the IDs of everything after it.

The Pro Git book

- Snapshots, Not Differences ✓
- Nearly Every Operation Is Local (*Sorry non-locality*) ✓
- Git Has Integrity (*Not like us*) ✓
- Git Generally Only Adds Data ✓
- The Three States (Working directory, Staging area, Repository) ✓
- Git is cool ✓

The challenge

The challenge - Goal

The goal is to find a string that looks like this

```
${CTF}NIPDt9MI0itWJD3FDBvxUBV7RQ9VNa
```

(it's start with `${CTF}` and the rest is (almost) random).

You only have to use git commands. You might also use the `cat` command to print the content of a file.

If you need authentication at some point, everything will be already set in place so you don't have to enter any password.

You can now find the useful code here: <https://github.com/nanoy42/git-ctf>

Get the source of this presentation here:

`https://github.com/nanoy42/git-presentation`

Origin of memes is written on the images.

Quotes where taken from the Git Pro Book.

The presentation uses the metropolis theme (<https://github.com/matze/mtheme>).

The rest of this presentation is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.



List of useful commands

List of useful commands

- `git commit`
- `git add`
- `git init`
- `git log`
- `git remote`
- `git blame`
- `git push`
- `git pull`
- `git fetch`
- `git merge`
- `git rebase`
- `git revert`
- `git reset`
- `git mv`
- `git rm`
- `git stash`