November 2023

Brief and practical crash course on:

Git & GitHub

What is Git?

Git is a free and open-source (distributed) Version Control System, originally authored by Linus Torvalds, the same creator of the Linux kernel. Git has become the most popular distributed VCS as it focuses on speed, data integrity and it supports distributed non-linear workflows.

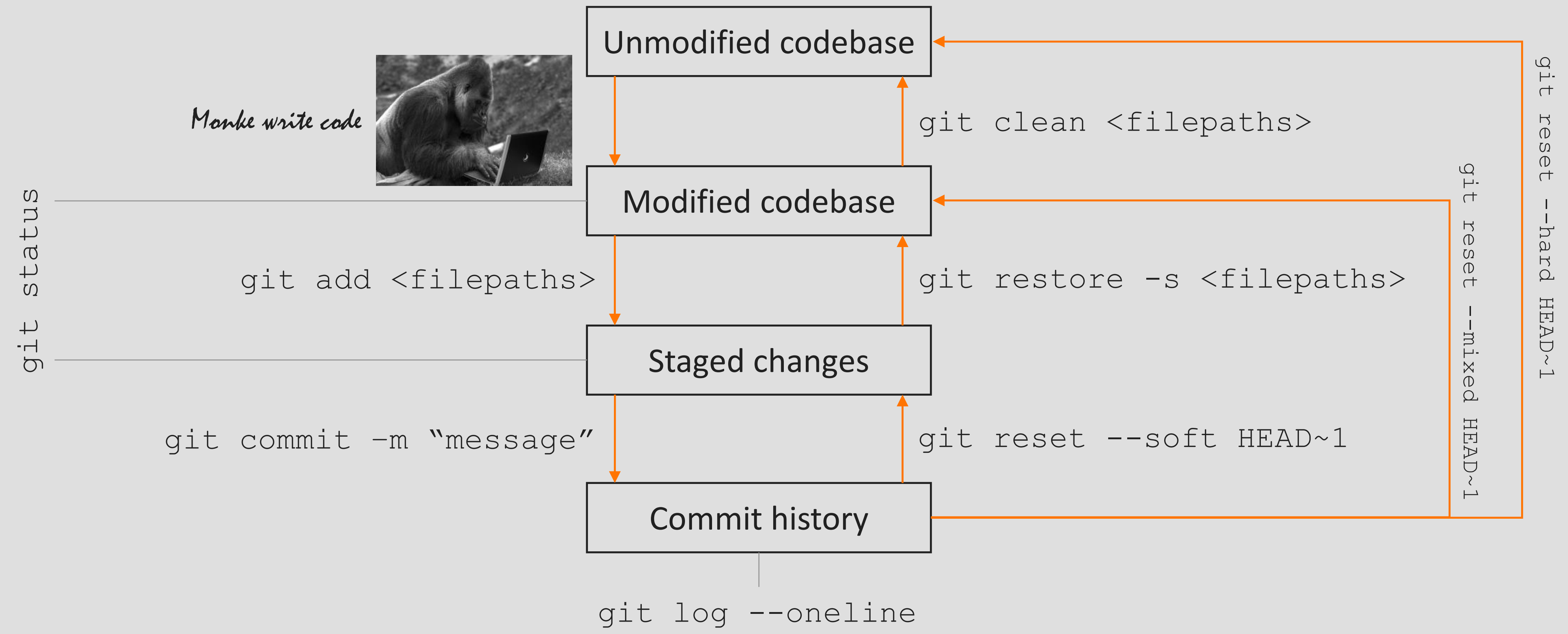
What is a Version Control System?

In software engineering, a VCS is a system responsible for managing changes computer programs, documents, or any sort of text-based information. A bit too abstract, so…

Why do we need a VCS?

- Tracking & review
- Data integrity
- Collaboration

Git 101



Monke write code

Your shift

1. Create a directory, switch to it, and initialise it as a local repository by running `git init`
2. Create a new file and run `git status`
3. Stage that file and rerun `git status`
4. Commit the staged changes with a message and see the git commit logs
5. Add & modify various files
6. Try to stage and un-stage them
7. Accumulate a bunch of commits and play around with the `reset` command

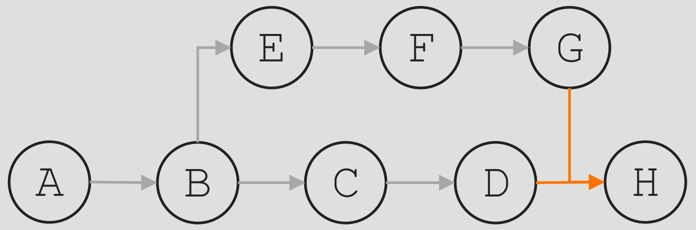
Real world tips

- Keep the commits' size small: fewer files ⇒ better tracking
- Give sensible names to commits and follow the team's naming convention
- Be careful with `git reset`: it is a powerful, and easily disruptive, command
- Not all files in your directory need version control ⟶ `.gitignore`

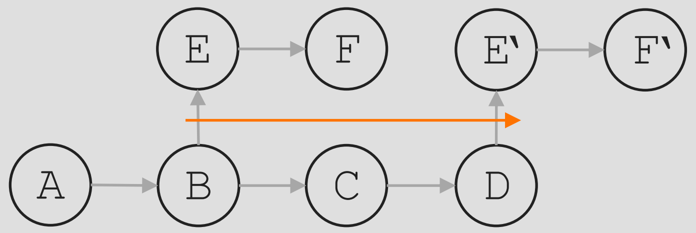
Where is the non-linear workflow?

Short answer: Branches
Long answer: Every software is made of various parts, that can be implemented/refactored/fixed in parallel.

- Each branch represent the same software, but with different features/bugfixes/implementations
- One always work on a specific branch at a time ⟶ `git checkout <name>`
- Commit histories of different branches are independent
- Every branch is created starting from a commit of another branch ⟶ `git branch <name>`
- When initializing, git silently creates the main branch, usually called `master` or `main`
- The initial commit of any branch can be moved forward along the original branch ⟶ `rebase`
- Committed changes of a branch can be incorporated into any other ⟶ `merge`



```
git checkout master
git merge topic
```



```
git rebase master topic
```

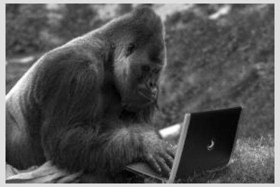
Non-linear remarks

- When the original branch received no commits, merging is done through fast-forwarding
- If different branches modified the same files, merging/rebasing requires manual conflicts resolution
- Use sensible names for branches too: a good practice is to categorize them as if they were pathnames
  e.g. `feat/frequency-change` `refactor/layout` `fix/ode-out-of-bounds`

Your shift

1. Create and switch to a new branch with `git checkout -b <name>`
2. Commit something on the new branch
3. Switch to the old branch (`git checkout main`) and merge the new branch (`git merge <name>`)
4. Create different files on both branches, commit, and merge
5. Modify the same file on both branches, commit, and merge
6. Repeat steps 4 and 5, but this time rebase instead of merging

Monke resolve conflit
Monke be angy



Where is the distributed workflow?

Git is distributed in the sense that the entire codebase – including its fully history – is mirrored on every developer's machine. Compared to centralized systems, it has many advantages, most notably:

- It allows developers to work entirely offline, as you did up until now
- Common operations – such as commits, viewing history, reverting changes – are faster
- It allows various development models to be used

But who can host a copy of the codebase always available for everyone? GitHub!

What is GitHub?

It is a cloud-based service for software development which provides the distributed version control of Git and many other useful features related to storage and management of code:

- Access control
- Bug tracking & feature request ⟶ issues tab
- Continuous integration, delivery, deployment ⟶ GitHub actions
- Documentation ⟶ readme files & wiki

Git + GitHub workflow

- Before anything, tell Git your GitHub user:
  1. Run `git config --global user.name “<your GitHub username>”`
  2. Run `git config --global user.email “<your GitHub user email>”`
  3. Create a (classical) personal access token ⟶ guide
  4. When prompted to login use your personal access token
- To start working locally, clone a remote repository:
  1. Go to the GitHub repository of your choice and copy the git HTTPS URL
  2. Run `git clone <HTTPS URL> <local directory name>`
- To contribute to a repository:
  1. Pull the changes from the remote repository: `git fetch` and `git pull`
  2. Code, stage, and commit local changes
  3. Push them to the remote repository: `git push`
- When pushing a newly created local branch, you will have to specify the remote branch
  - The first time run: `git push --set-upstream origin <branch name>`
  - Keep local and remote branches homonymous for ease of understanding

Your shift

1. Clone the playground repository
2. Create a branch with a playful name, add a file or modify the readme, commit and push
3. Push another commit to your branch
4. Go one commit back locally (use `git reset`)
5. Try to push and if you fail force push i.e. run `git push --force`
6. Try to commit and push something to the `main` branch

Night shift in pairs

1. After fetching, Alice switches to Bob's branch and vice versa
2. Both modify the readme on the same line, push, and then switch back to their respective branches
3. Alice commits and pushes a change to her branch
4. Bob pulls and then commits and pushes some changes on his branch
5. Bob further modifies the readme but "forgets" to commit or push
6. Alice switches again to Bob's branch, modifies the readme and pushes
7. Bob comes back and before doing anything pulls

Warnings

- Always pull before working
- Don’t leave your local repository with un-pushed or un-committed changes
- Some branches are protected and can only be changed by means of pull requests

Security

We must prevent unauthorized user from making:

- Irrevocable damage to our code
- Un-reviewed changes to the “important” branches

For this reason, GitHub offers many security strategies. In our case:

- Members are divided into teams, each with write access only to specific repositories
- Creation and modification of repositories, new member invitations can be performed only by the IT lead
- Addition to and removal of members from a team can be done also by leads
- There must be a protected “release” branch to ensure availability, integrity, authenticity