One-page summary for Git

Yunzhe Li @UC Berkeley
liyunzhe.jonas@berkeley.edu

Vision

As a technical manager and a team leader, my main challenge is making sure that every engineer involved is aligned on a shared version.**[Challenge]** To do that, it is essential to have a version control system**[How to solve it]** where each engineer can store and share the code they create.**[Advantage-1]** This lets us track different revisions, roll back problematic changes, and work together effectively**[Advantage-2]**.

Contributions

Though files can be differentiated automatically by Linux commands, Git is a powerful tool for version control on large-scale changes in files. Basic operations like creating or cloning a repository, adding code, checking the status, and committing changes are well-introduced in the local repo. In addition, advanced interactions like deleting, renaming, ignoring, undoing in different stages, and branching and merging are emphasized. Then we push changes from our local repos to the remote one where we can fetch changes, solve conflicts, and most importantly, collaborate with our peers through GitHub. Finally, several tips including pull requests, issue trackers, and continuous integration make it all come alive.

Steps

Before version control

Before the version control system, we used Linux commands to apply differences between collaborators in files. The dash u means unified.

- `diff -u old_file new_file > filePatch.diff`
- `patch target_file < filePatch.diff`

Basic git workflow

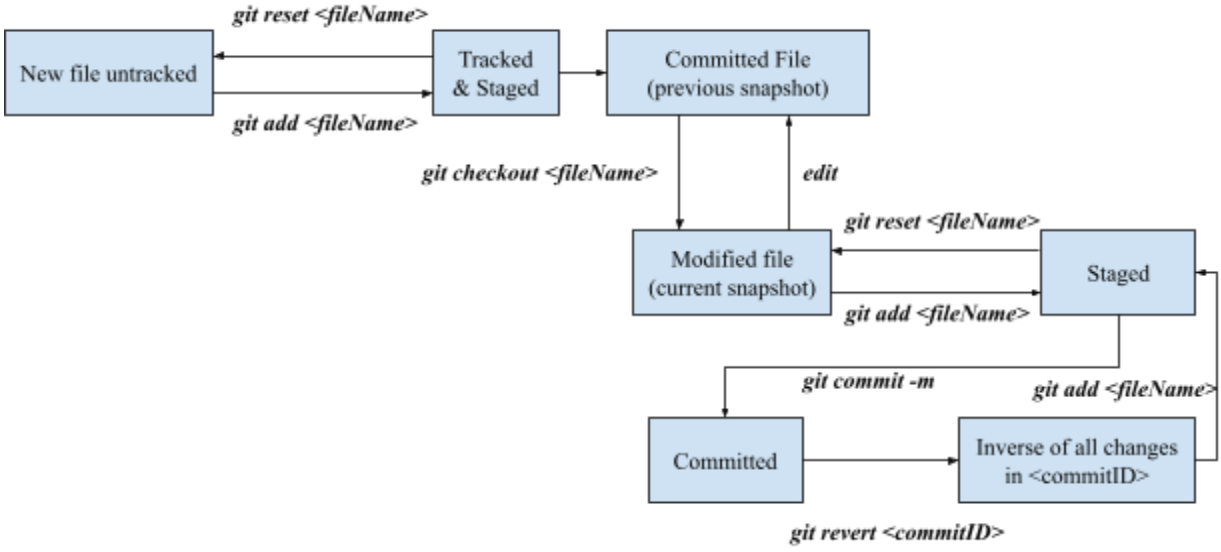
In large projects that update frequently and work with many people, it would be frustrating when there's a bug detected and it cannot be solved in a short period. The Git version control system served as a centralized location called a repository for collaborators to track changes in project files. The standard process is:

1. Run ***sudo apt update*** to make sure you have a fresh index of the packages available.

2. Run ***sudo apt install git***
3. Create a directory, then ***git init*** or ***git clone <url>*** to set a local repo. Then the directory will have a working tree and a Git directory to save the change history.
4. Work on the files which are in the working tree. Use ***git diff*** to show unstaged changes. Use ***git rm*** to delete one file and ***git mv*** to rename or move one file.
5. Use ***git add <fileName>*** to mark the files that need to be tracked and they will be staged in the staging area. If we want to ignore some files, create a **.gitignore** file and add target suffixes.
6. Use ***git commit -m <messages for this commit>*** to commit changes in staged files. Changes will be taken snapshots and safely stored in the database that lives in the Git directory. Shortcut: for those tracked files(not newly added ones) that have slight changes, use ***git commit -a -m <messages>*** to skip the staging step.
7. Run ***git status*** to check the current status in the staging area and ***git log*** to check the commit history in this repo. The ***git log -p*** command can show actual lines in changes. If we want to check one specific commit, use ***git show <commitID>***.

Undoing

One of the most crucial functions of Git is its ability to revert what we have wrongly made. The **HEAD** representing the currently checked-out snapshot is used to undo operations and switch between several branches. The standard workflow is as follows. In your local repo, if you want to overwrite one commit, you can use ***git commit --amend*** to take whatever is currently in the staging area and run the git commit workflow to overwrite the previous commit.



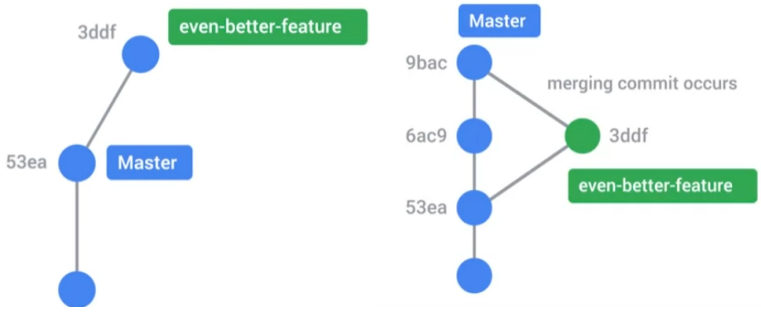
Git revert is to create a new commit that contains the inverse of the changes in the specific commit identified by commit ID.

Branching and Merging

A ***Branch*** is a pointer to a particular commit. The default branch is called Master. Experiments can be worked on separate branches in a normal development workflow without disrupting the most current working state on the Master branch.

- Use ***git branch*** to list all branches in the local repo.
- Use ***git branch <branch-name>*** to create a new branch.
- Use ***git checkout <branch-name>*** to switch the branch.
- Use ***git checkout -b <branch-name>*** to create a new branch and automatically switch to it.

The git merge command is used to take the independent snapshots and history of one git branch and tangle them into another. There are two merging algorithms: fast forward for non-divergence merge and 3-way mering for divergence merge.



This happens when there's one or more commits are made after the point when both branches split(53ea in this example). Git will try to figure out how to merge by tying the branch histories together with a new commit and merge snapshots at the most recent common ancestor(53ea in this example).

1. Use ***git merge <toBeMergedBranch>***, then a merge conflict may occur if changes are made in the same part of the same file.
2. After re-editing the conflict file, we need a new commit that contains the staged merged file to resolve the conflict.
3. Use the ***git log --graph --oneline*** to see the graphic path of the merge.
4. If you want to throw the merge away, use the ***git merge --abort*** to stop.

Remote Repo

1. Use git clone <url>
2. Check configuration. Use ***git remote -v*** to check the configuration of the remote repo. Use ***git remote show origin*** to get more information. Use ***git branch -r*** to check the currently tracked branches in the remote branch. When using ***git status***, the

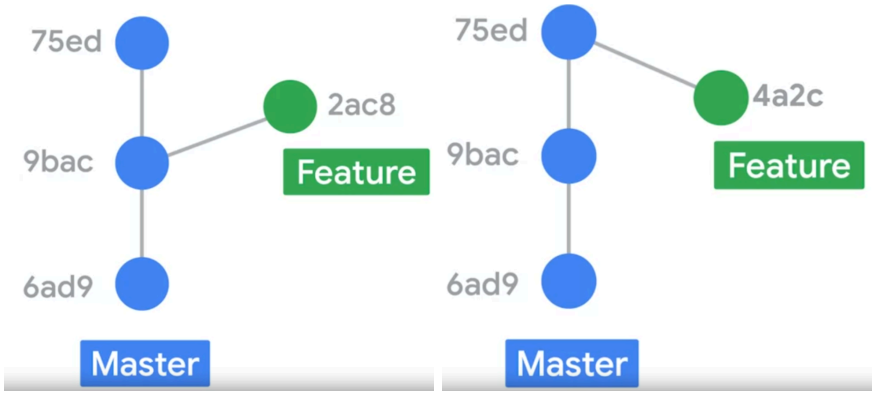
"origin/master" means the master branch in the remote repo called origin, has the same commits as our local master branch.

3. Sync data. Use ***git pull*** to update data and merge to our current one, while ***git fetch*** just updates data but does not merge.
4. After editing our files locally, execute the commit workflow and use ***git push*** to update changes to the remote.
5. If we want to create a local branch for it, we can use ***git checkout <branchName>***.
6. If we want to **fetch** the contents of all remote branches without automatically merging any contents into the local branches, we can call ***git remote update***.

Conflict. When there's conflict in ***git pull*** process, use ***git log --graph --oneline --all*** to see the log. Use ***git log -p origin/master*** to see the changes made by our colleagues. Open the file with conflict, edit manually, add them, and commit them.

New branch. When we push a new local branch to the remote repo, use the ***git push -u origin <branchName>***. The -u flag is used to create the branch upstream which is another way of referring to remote repos.

Rebasing. To make debugging easier and prevent three-way merges by transferring the completed work from one branch to another, the command ***git rebase branchName*** moves the current branch on top of the branchName **IN OUR LOCAL REPO**.



After rebasing, we need to merge the refactor onto the main branch in a fast-forward way using the ***git merge <branchName>*** command. Then use ***git branch -d <branchName>*** and ***git push --delete origin <branchName>*** to delete the refactor branch locally and remotely. And finally, git push to the remote repo. **HEADS UP: This operation does not make any changes to the remote refactor branch, just updating the master branch after rebasing locally.**

The fetch-rebase-push workflow is: first, use ***git fetch*** to download changes but not merge in a three-way. Then use ***git rebase***. Last, ***git push***.

Best Practices for Collaboration

1. Always ***synchronize your branches*** before starting any work on your own.
2. ***Avoid*** having ***very large changes*** that modify a lot of different things.
3. When working on a ***big change***, it makes sense to have a ***separate feature branch***.
4. ***Regularly merge*** changes made on the master branch ***back onto*** the ***feature branch*** to make the final merge easier.
5. Have the ***latest version*** of the project in the ***master branch***, and the ***stable version*** of the project on a ***separate branch***.
6. You ***shouldn't rebase changes*** that have been pushed to ***remote repos***.
7. Having good commit messages is important.

>> My first step is to work backward and disable everything I've done and then see if the source still works, then I slowly add pieces of code until I hit the problem. That usually gets me through the tough times and has definitely highlighted some weird occurrences.

Pull Request & Collaborating

The Pull Request is a commit or series of commits that you send to the owner of the repo so that they incorporate it into their tree. This is a typical way of working on GitHub, because, in most projects, only a few people have commit rights to the repo. But anybody can suggest patches, bug fixes, or even new features by sending pull requests that people with commit access can apply.

The typical pull request workflow on GitHub

1. Fork the repo on the GitHub
2. Git clone to the local
3. Create a new branch using ***git checkout -b branchName***
4. Modify as you wish
5. Stage and commit changes
6. Create a remote repo by ***git push -u origin branchName***
7. After you send a pull request, the maintainer may comment on them to request some missing documents or improvements in coding style. Then you need to update your PR. The GitHub will gather commits in the same branch into the same PR
8. To squash two commits into one, use the ***git rebase -i main*** command where -i flag means interactive. Instead of merging to divergence like previously what we do, we just use ***git push -f*** to force the git to push.
9. Leave a message that "All comments are resolved, please take another look." when you solve all comments and then push it.
10. If you are a project maintainer, it's important that you reply promptly to pull requests and don't let them stagnate. A common strategy for active software projects is to use an issue

tracker. Once you solve the issue identified as #4, you can leave a message of **“Closes #4**” when committing it. Then Git will automatically close the issue.

The continuous integration system builds and tests our code every time there’s a change. Once we have our code automatically built and tested, the next automation step is **continuous deployment**, which is sometimes called **continuous delivery(CI/CD)**.

In terms of source control, you're **downstream** when you copy (clone, checkout, etc) from a repository. Information is flowed **downstream** to you. When you make changes, you usually want to send them back **upstream** so they make it into that repository so that everyone pulling from the same source is working with all the same changes. You want to get your changes into the **main** project so you're not tracking divergent lines of development.
Setting the **upstream** for a fork you have created using the following command:

git remote add upstream https://github.com/[git-username]/it-cert-automation-practice.git