Git hub

20 August 2023 23:58

Git Commands

Setup : Set the name and email that will be attached to the commits and tags.

- **git config --global user.name "Sadat"**
- **git config --global user.email "sadat3332@gmail.com**

Starting a project

- **git init <directory>** (initializes repository)
- **git clone <URL>**

Common Actions :
- Reordering commits : rearranging the order of commits.
- Editing commits : modifying the commit message, content.
- Squashing commits : combining multiple commits into a single commit
- Dropping commits : excluding certain commits

Changes

- **git add <file>** (Add a file to staging)
- **git add .** (Stage all files)
- **git commit -m "this is a message"** (commit all staged files to git)

Branches

- **git branch** (Lists all local branches, add -r flag to show all remote branches , -a flag for all branches)
- **git branch <new-branch>** (Creates a new branch)
- **git checkout <branch>** (switch to a branch and update the working directory)
- **git checkout -b <new-branch>** (Create a new branch and switch to it)
- **git branch -d <branch>** (Delete a merged Branch)
- **git branch -D <branch>** (Delete a branch whether merged or not)
- **git tag <tag-name>** (Add a tag to current commit)

{

git log --oneline (commit id in SHA)

git tag <tag-name> <commit-SHA-1>

git push origin <tag-name>

}

Git useful commands:

- **Git config --global alias.<alias-name> <git-command-it>**(configuring alias name)
- **Git config --global core.editor <editor>** (configure editor)
- **Git config --global --edit** (config in editor)

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.up rebase
git config --global alias.ci commit
```

Editors names :

VS code : "code"
Sublime text : "subl"
Notepad++ : "notepad++"
Notepad : "notepad"
Emacs : "emacs"
Wordpad : "wordpad"

Merging

git checkout <branch1> **(Merge branch2 into branch1)**
git merge <branch2>

git merge --squash branch2 (Merge and squash all commits into one new commit)

```
A --- B --- C --- F (main: Merge feature)
                \   /
                 D -- (feature: Deleted file X)
                  \
                   E -- (feature: Deleted file Y)
```

Merge :

Git merge <branch-name>

Fast forward merge: The receiving branch did not get any changes since the two branches diverged. The receiving branch still points to the last commit before the other branch diverged. In this case, Git moves the branch pointer of the receiving branch forward as shown in Figure 5. Because there is nothing to do besides moving the branch pointer forward, Git calls this a fast forward merge.

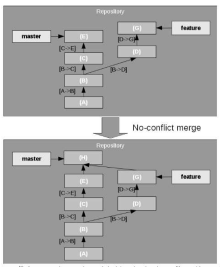
Rebasing

Rebase feature branch onto main to incorporate changes made to main. Prevents unnecessary merge commits into feature keeping history clean.

Shows log in form of graph
Git log --graph --oneline

- **git checkout test**
- **git rebase main**
- **git rebase -i main** **(interactively clean up a branches commits before rebasing onto main)**

No-conflict merge: There are changes in both branches but they do not conflict. This happens, for example, if the changes in both branches affect different files. Git can automatically apply all changes from the other branch into the receiving branch, and create a new commit with these changes included.



- **Conflicting merge: There are changes in both branches, but they conflict. In this case, the conflicting result is left in the working directory for the user to fix and commit, or to abort the merge with git merge --abort.**

Undoing things

- **git mv <existing path> <new path>** (Move / or rename a file)
- **git rm <file>** (remove a file from working directory and staging area)
- **git rm --cached <file>** (remove from staging area only)
- **git checkout <commit_ID>** (view previous commits)
- **git revert <commit_ID>** (create a new commit , reverting the changes from a specified commit)
- **git reset <commit_ID>** (go back to a previous commit and delete all commits ahead of it , add --hard flag to also delete workspace changes)

Reviewing Repo:

- **git status** (list new or modifies files not yet committed)
- **git log --oneline** (list commit history , with SHA-1 ID)
- **git diff** (show changes to unstaged files, for changes to staged files add --cached)
- **git diff commit1_ID commit2_ID** (show changes between two commits)

Stashing

- When Branches with conflicts are switched , changes that aren't committed would be lost . However If we want to save the changes without committing it ,we use stash.

- **Git stash** (stores modified and staged changes , for untracked files -u flag is added , -a flag for untracked and ignored files)
- **Git stash save "comment"** (stash the modified changes with description)
- **Git stash list** (lists all stashes)
- **Git stash apply** (Reapply the stash without deleting it)
- **Git stash pop stash@{2}** (reapply the stash at index 2, then delete it from stash list)
- **Git stash show stash@{1}** (show the diff summary of stash 1)
- **Git stash drop stash@{1}** (Delete stash at index 1. omit stash@{n} to delete last stash made)
- **Git stash clear** (Delete all stashes)

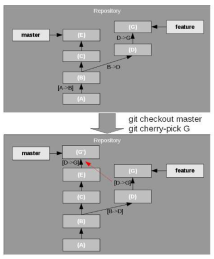
Git hub commands

- **Git remote** (view all remote connections, -v flag to check URLS)
- **Git remote remove <alias>** (Remove a connection)
- **Git remote rename <old> <new>** (Rename a connection)
- **Git fetch <alias>** fetch all branches from remote repo without merging)
- **Git fetch <alias> <branch>** (fetch a specific branch)
- **Git pull** (Fetch the remote repo's copy of current branch , then merge)
- **Git pull --rebase <alias>** (rebase your local changes onto top of new changes made to the remote repo)
- **Git push** (update local content to remote repo)
- **Git push <alias> <branch>** (Upload to a branch)

To clean the untracked files
Git clean -f , (-f == force)

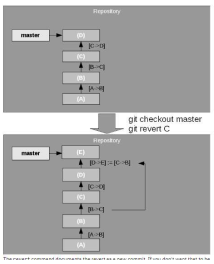
Cherry picking

Imagine you are now working on a feature, and have developed some change that should be put into your master development immediately. This could be a bug fix, or a cool feature but you don't want to merge or rebase the branches yet. Git allows to copy a change set from one branch to another by using the cherry pick feature.



Revert

The revert command rolls back one or more patch sets on the working directory, then creates a new commit on the result. revert is almost the reverse of a cherry pick. See Figure 9 for an example.



A new commit is generated reverting the changes and the history of commits is maintained.

Commit IDs:
In Git, each commit is uniquely identified by a long hexadecimal string known as the commit ID or hash. This hash is generated based on the contents of the commit, including the snapshot of the project's files, commit metadata (such as author, timestamp, and commit message), and the IDs of its parent commits (if applicable). Because this hash is derived from the commit's content, even a small change in any part of the commit will result in a completely different hash.

Data Structure: Directed Acyclic Graph (DAG):
The commit IDs play a crucial role in creating a data structure called a directed acyclic graph (DAG) in Git. This graph represents the commit history of a repository. Each commit is a node in the graph, and the edges (arrows) between nodes indicate the parent-child relationships between commits.

Creating a New Commit:
When you create a new commit, Git takes a snapshot of the current state of your project's files. It then generates a unique commit ID based on the content of this snapshot, along with metadata. The new commit points back to the previous commit as its parent. If you're on a branch, the new commit becomes the latest commit in that branch.

Parent-Child Relationships:
The parent-child relationships between commits form the DAG. Each commit has one or more parents (usually one, but more in the case of merge commits). This structure captures the history of changes and the order in which commits were created.

Navigating the DAG:
Git can navigate the commit history using the parent-child relationships. Starting from the most recent commit, Git can follow the parents to move back in time, effectively tracing the entire history of changes. This traversal allows Git to reconstruct the state of the project at any commit.

Branches and References:
Branches in Git are simply references to specific commits. When you create a new branch, Git creates a new reference pointing to the same commit as the current branch. As you make new commits on the branch, the branch reference moves forward to point to the latest commit.

Merging and Merge Commits:
When you merge changes from one branch into another, Git creates a new commit called a merge commit. This commit has two (or more) parent commits, representing the commits being merged. The merge commit combines the changes from both branches and resolves conflicts if any