Introduction to Git Commands

Krishna Singha

23CS10034

GIT STASH

GIT STASH

git stash is a command used to temporarily save changes that you've made to your working directory. This can be useful when you need to switch to a different branch or work on a different task without committing your changes.

Here's how it works:-

Stash Changes: You use the command *git stash* to stash your changes. This command takes the current state of your working directory and index and saves it on a stack of unfinished changes.

Continue Working: After stashing your changes, you can continue working on something else, switch branches, or do whatever you need to do.

Apply Stashed Changes: When you're ready to reapply your changes, you can use *git stash apply* to reapply the most recent stash to your working directory.

Remove Stashed Changes: You can also remove stashed changes from the stack using *git stash drop*.

List Stashed Changes: To see a list of stashed changes, you can use *git stash list*.

Overall, *git stash* is handy for situations where you're not ready to commit your changes but need to work on something else temporarily. It allows you to store your work-in-progress without cluttering your commit history.

GIT BISECT

GIT BISECT

git bisect is a command that helps you find the commit that introduced a bug or regression in your codebase. It operates on the principle of binary search.

Here's how it works:-

Start Bisecting: You start by telling Git that you want to begin the bisect process using *git bisect start*.

Identify Good and Bad States: You need to identify a "good" state (a commit where the bug didn't exist) and a "bad" state (a commit where the bug does exist). You can mark these states using *git bisect good <commit>* and git bisect bad <commit>, respectively. If you're currently on a bad commit, you can simply use git bisect bad without specifying a commit.

Git Does the Work: Git will then use a binary search algorithm to check out commits between the good and bad states, allowing you to test whether the bug exists in each commit.

Test and Repeat: After checking out a commit, you test your code to see if the bug exists. If the bug is present, you mark the commit as bad (*git bisect bad*); if not, you mark it as good (git bisect good). Git will then automatically select the next commit for testing based on your feedback.

Repeat Until Found: You continue this process of testing and marking commits until Git identifies the exact commit that introduced the bug.

Finish Bisecting: Once Git finds the problem commit, it will display the commit hash. You can then exit the bisect process using git bisect reset.

Overall, Git bisect is a powerful tool for efficiently identify when and where bugs were introduced in your codebase, which can really help to make debugging faster and easier.

GIT REFLOG

GIT REFLOG

git reflog is a command that displays a log of reference updates in your repository, including when branches were created or deleted, when commits were made, and when the HEAD moved. It stands for "reference logs".

Here's how it works:-

Viewing the Reflog: You can utilise *git reflog* to show a list of recent actions in order and the commit codes linked with them.

Understanding the Output: The output of *git reflog* typically includes columns like the commit hash, the action performed (e.g., commit, reset, merge), the HEAD position, and a description of the action.

Recovering Lost Commits: One common use of *git reflog* is to recover lost commits or changes. If you accidentally reset your branch or deleted a commit, you can often find the lost commit hash in the reflog and then use *git checkout* or *git reset* to return to that state.

Diagnosing Issues: *git reflog* is also helpful for diagnosing issues related to branch history, identifying when changes were made, and understanding the sequence of actions that led to the current state of the repository.

Overall, *git reflog* provides a detailed history of reference updates in your Git repository, making it a valuable tool for understanding and managing your project's history.

GIT DIFF

GIT DIFF

git diff is a command that shows the differences between different states of your repository. It's useful for understanding what changes have been made to your files since the last commit, comparing different branches or commits, and identifying modifications before committing them.

Here's how it works:

Comparing Working Directory with Staging Area: If you run *git diff* without any arguments, it will show the differences between the files in your working directory and the files that have been staged (added to the index) but not yet committed.

Comparing Staging Area with Repository: To compare the changes that have been staged with the latest commit in the repository, you can use *git diff —staged* or *git diff —cached*.

Comparing Between Commits: You can also compare different commits by specifying their commit hashes or branch names. For example, *git diff commit-1 commit-2* will show the differences between the two specified commits.

Viewing Unified or Context Diffs: By default, *git diff* shows a unified diff, which includes the added and removed lines of code along with some context.

Comparing Specific Files: If you only want to see the differences in specific files, you can specify their paths after the *git diff* command.

Overall, *git diff* is a versatile tool for examining changes in your repository and understanding the differences between various states of your project.

GIT SWITCH

GIT SWITCH

git switch is a command introduced in Git 2.23, primarily used to switch between branches or to create a new branch and switch to it in a single step. It's a simpler way to do things compared to using both *git checkout* and *git branch* commands together.

Here's how it works:-

Switching Branches: If you want to switch to an existing branch, you can use *git switch <branch_name>*. This command will update your working directory to match the state of the specified branch.

Creating and Switching to a New Branch: If you want to create a new branch and switch to it immediately, you can use *git switch -c <new_branch_name>*. This command will create a new branch with the specified name and then switch to it.

Additional Options: *git switch* also offers extra choices, like picking a particular commit or tag to move to, forcefully switching to a branch (which discards local changes), and starting a new history separate from any existing branches by creating an orphan branch.

Overall, *git switch* simplifies the process of switching between branches and creating new branches, making branch management more intuitive and efficient.

GIT REBASE

GIT REBASE

git rebase is a command used to integrate changes from one branch into another by reapplying commits on top of another base branch. It's a way to maintain a cleaner, linear project history compared to merging.

Here's how it works:

Starting a Rebase: You typically start a rebase by checking out the branch you want to rebase onto (the base branch), and then running *git rebase <branch_to_rebase>*. This command tells Git to reapply the commits from *<branch_to_rebase>* on top of the current branch.

Reapplying Commits: Git will go through each commit in *<branch_to_rebase>*, calculate the differences, and then reapply those changes on top of the base branch one by one.

Resolving Conflicts: If there are any conflicts during the rebase process, Git will pause and allow you to resolve them manually. After resolving conflicts, you continue the rebase process by running *git rebase --continue*.

Aborting a Rebase: If you encounter any issues or conflicts that you cannot resolve, you can abort the rebase with *git rebase --abort*. This returns your repository to the state it was in before you started the rebase.

Completing the Rebase: Once all commits have been successfully reapplied, the rebase is complete, and your branch will now contain the commits from *<branch_to_rebase>* on top of the base branch.

Overall, *git rebase* is often used to maintain a cleaner project history, especially in collaboration with others, as it can result in a linear sequence of commits. However, it's essential to use rebase with caution, especially when working on shared branches, as it rewrites commit history, which can cause issues for collaborators.

GIT CHERRY-PICK

GIT CHERRY-PICK

git cherry-pick is a command that allows you to pick specific commits from one branch and apply them onto another branch. It's useful when you want to bring in individual commits from one branch to another without merging the entire branch.

Here's how it works:

Identify the Commit: You start by identifying the commit(s) you want to apply to another branch. You can find the commit hash either in the commit log or using tools like *git log*.

Switch to the Target Branch: Make sure you're on the branch where you want to apply the selected commit(s). You can switch branches using *git checkout <target_branch>*.

Cherry-pick the Commit: Use *git cherry-pick <commit_hash>* to apply the specified commit onto the current branch. You can cherry-pick multiple commits by specifying their respective commit hashes.

Resolve Conflicts (if any): If there are any conflicts between the changes introduced by the cherry-picked commit and the current state of the branch, you'll need to resolve them manually.

Complete the Cherry-pick: After resolving conflicts, stage the changes using *git add* and continue the cherry-pick operation by running *git cherry-pick --continue*. If you encounter issues that you can't resolve, you can abort the cherry-pick using *git cherry-pick --abort*.

Repeat as Needed: You can repeat the cherry-pick process to bring in additional commits from the source branch to the target branch.

Overall, *git cherry-pick* is handy for selectively applying changes from one branch to another, especially when merging the entire branch isn't necessary or desirable.

THANK YOU