The Github SAGA

By Renusree Rapolu

Git-Stash

What is Git Stash :

Git stash is basically used when you want to put aside any uncommitted changes in the current working directory and goes back to whatever we want to work on , it basically is like holding pen

This helps developers to switch branches , work on other directories and collaborate with others without affecting the current work

Why :

We might want to switch to another repository or come back to this later

How Does it Work:

it stores in stacks , when you apply the command it stores it in a safe place for us

You can also add a message to describe your stash using `git stash save "Your message here"`, which can be helpful for later reference

Viewing Stashes:

You can view a list of all your stashes by running `git stash list`.

Each stash is assigned a unique identifier (stash@{N}), which you can use to reference it later.

Applying Stashed Changes:

To apply the most recent stash, you can use `git stash apply`.

This will reapply the stashed changes to your working directory, leaving the stash intact.

If you have multiple stashes, you can specify which one to apply using `git stash apply stash@{N}`.

Pop vs Apply:

There are two main ways to apply stashed changes: `git stash apply` and `git stash pop`.

`git stash apply` will apply the changes but leave the stash intact, allowing you to apply it again later if needed.

`git stash pop`, on the other hand, will apply the changes and remove the stash from the stack.

Dropping Stashes:

If you no longer need a particular stash, you can drop it from the stack using `git stash drop stash@{N}` and it will permanently delete the stash and for clearing all the stashes from the stack use GIT STASH CLEAR

Applying Partial Stashes:

Sometimes you might only want to apply certain changes from a stash so you can do this by first applying the stash using `git stash apply`, and then using `git add` and `git commit` to selectively commit the changes you want to keep.

Git bisect

Introduction to Git Bisect

- Git bisect is a powerful tool in Git used for pinpointing the commit that introduced a bug or regression in your codebase.
- It automates the process of finding the exact commit responsible for a bug by performing a binary search through the commit history.

Why Use Git Bisect?

- When debugging a complex issue, it can be challenging to manually search through the commit history to find the culprit.
- Git bisect automates this process, saving time and effort by quickly narrowing down the range of commits that may contain the bug.

How Does Git Bisect Work?

- Git bisect uses a binary search algorithm to efficiently locate the commit that introduced a bug.
- It requires you to specify a "good" commit (where the bug is not present) and a "bad" commit (where the bug is observed).
- Git bisect then systematically tests commits in between to determine the first "bad" commit.

Setting Up Git Bisect

- To start using Git bisect, first, identify a "good" commit (where the bug is not present) and a "bad" commit (where the bug is observed).
- Use `git bisect start` to initialize the bisect process.
- Use `git bisect good <commit>` to mark the "good" commit.
- Use `git bisect bad <commit>` to mark the "bad" commit.

Automating the Process

- Once you've marked the "good" and "bad" commits, Git bisect will automatically begin testing commits in between using a binary search algorithm.
- It will check out commits one by one and prompt you to test each commit to determine if the bug is present.
- Based on your feedback, Git bisect will continue bisecting until it identifies the first "bad" commit.

Finding the First Bad Commit

- After testing several commits, Git bisect will eventually narrow down the range to a single commit—the first commit where the bug was introduced.
- This commit is identified as the first "bad" commit, and Git bisect will provide you with its hash.

Completing the Bisect Process

- Once Git bisect identifies the first bad commit, you can use `git bisect reset` to exit the bisect process and return to the original state.
- This command resets the repository to its state before the bisect started and checks out the original branch.

Git reflog

Introduction to Git Reflog

- Git reflog, short for reference log, is a built-in tool in Git that records the history of references (e.g., branches, commits, HEAD) in your repository.
- It serves as a safety net, allowing you to recover lost commits, branches, or undo destructive operations even if they are not referenced by any branch.

Why Use Git Reflog?

- Git reflog is particularly useful when you've accidentally deleted a branch, reset to an unintended commit, or made other changes that are no longer visible in the commit history.
- It provides a comprehensive record of all reference changes, enabling you to navigate and recover from mistakes effectively.

How Does Git Reflog Work?

- Git reflog works by maintaining a chronological log of all reference updates in your repository.
- Each entry in the reflog contains information such as the commit hash before and after the reference change, the action performed, and a timestamp.
- This log is stored locally within the `.git` directory of your repository and is not shared with remote repositories.

Viewing the Reflog

- To view the reflog, use the command `git reflog`.
- This will display a list of reference changes, including commits, branch creations/deletions, resets, merges, and other operations.
- Each entry in the reflog is assigned a unique identifier (e.g., HEAD@{N}), which you can use to reference specific entries.

Recovering Lost Commits

- One common use case for Git reflog is to recover lost commits.
- If you've accidentally reset to a previous commit or deleted a branch, you can use the reflog to find the commit hash before the change and reset your branch back to that state.

Restoring Deleted Branches

- If you've accidentally deleted a branch, the reflog can help you restore it.
- Find the entry in the reflog corresponding to the deletion of the branch and use `git checkout -b <branch_name> <commit_hash>` to recreate the branch at the desired commit.

Undoing Reset Operations

- If you've accidentally reset your branch to an unintended commit, the reflog can help you undo the reset.
- Find the entry in the reflog corresponding to the reset operation and use `git reset --hard <commit_hash>` to revert your branch back to that commit.

Recovering from Mistakes

- Git reflog provides a safety net for recovering from various mistakes and accidents in your Git workflow.
- By maintaining a detailed log of reference changes, it enables you to navigate through your repository's history with confidence.

Git diff

Introduction to Git Diff

- Git diff is a fundamental command in Git used to compare changes between different states of your codebase.
- It allows you to see the differences between commits, branches, or individual files, providing valuable insights into the evolution of your code.

Why Use Git Diff?

- Git diff is essential for understanding how your code has changed over time and identifying potential issues or conflicts.
- It helps you review modifications before committing, resolve merge conflicts, and track changes between different branches or commits.

How Does Git Diff Work?

- Git diff compares the contents of files in different states (e.g., working directory, staging area, commits) and highlights the differences.
- It analyzes the changes line by line, showing additions, deletions, and modifications using standard diff notation (+ for additions, - for deletions).
- Git diff can be used to compare changes between commits (`git diff <commit1> <commit2>`), between the working directory and staging area (`git diff --staged`), or between the working directory and the last commit (`git diff HEAD`).

Viewing Changes Between Commits

- To view the changes between two commits, use the command `git diff <commit1> <commit2>`.
- This command shows the differences between the specified commits, highlighting additions, deletions, and modifications.
- You can specify branch names, commit hashes, or relative references (e.g., HEAD~1, HEAD^) to compare different states of your repository.

Viewing Changes in the Working Directory

- To view changes between the working directory and the staging area, use `git diff`.
- This command compares the current state of files in your working directory with the last committed state, highlighting modifications that have not been staged.
- It helps you review changes before staging them for commit.

Viewing Changes in the Staging Area

- To view changes between the staging area and the last commit, use `git diff --staged` or `git diff --cached`.
- This command compares the changes that have been staged (i.e., added to the index) with the last committed state, highlighting modifications that will be included in the next commit.
- It helps you review staged changes before committing them.

Viewing Changes for Individual Files

- Git diff can also be used to view changes for individual files.
- Simply specify the filename or path to the file you want to compare, e.g., `git diff <file>` or `git diff -- <file>`.
- This command shows the differences between the specified file in the working directory and the last committed state.

Diff Options and Customization

- Git diff offers various options and customization features to tailor the output to your preferences.
- You can use options like `--color`, `--word-diff`, `--word-diff-regex`, `--stat`, and more to enhance readability and gain additional insights into the changes.
- Experiment with different options to find the best configuration for your workflow.

Git switch

Introduction to Git Switch

- Git Switch is a new command introduced in Git 2.23 to provide a safer and more intuitive way to switch between branches compared to `git checkout`.
- It simplifies the process of changing branches and reduces the risk of unintentional changes to the working directory.

Why Use Git Switch?

- Traditional branch switching with `git checkout` can sometimes lead to unintended consequences, such as inadvertently detaching HEAD or overwriting local changes.
- Git switch offers a more straightforward and error-resistant approach to branch switching, enhancing the overall Git user experience.

How Does Git Switch Work?

- Git switch simplifies branch switching by focusing solely on changing branches, without the additional functionality of `git checkout`.
- It checks whether the given branch exists and is safe to switch to before executing the operation, reducing the likelihood of errors.
- Git switch also provides clearer error messages and suggestions for resolving conflicts or issues.

Basic Usage of Git Switch

- To switch to an existing branch using Git switch, simply run `git switch <branch_name>`.
- This command checks out the specified branch, updating the working directory and HEAD pointer accordingly.
- Git switch will prevent the switch if there are uncommitted changes that would be overwritten by the operation.

Creating and Switching to a New Branch

- Git switch allows you to create and switch to a new branch in a single step using the `-c` or `--create` option.
- For example, `git switch -c <new_branch>` creates a new branch and immediately switches to it.
- This streamlines the process of branching off from the current state of the repository.

Safety Features of Git Switch

- Git switch includes safety features to prevent unintentional operations and protect the integrity of the repository.
- It checks whether switching branches would result in uncommitted changes being overwritten or conflicts arising from unstaged changes.
- Git switch also provides suggestions for resolving conflicts or issues encountered during branch switching.

Git Switch vs. Git Checkout

- While `git checkout` remains a versatile command with various functionalities, `git switch` is dedicated solely to branch switching.
- Git switch offers a more focused and streamlined approach, reducing the likelihood of errors and simplifying the user experience.

Compatibility and Adoption

- Git switch is compatible with Git 2.23 and later versions, ensuring broad support across different environments and platforms.
- It's recommended to adopt `git switch` in your Git workflow to leverage its benefits and improve branch management.

Git rebase

Introduction to Git Rebase

- Git rebase is a powerful command in Git used to reapply commits from one branch onto another, effectively rewriting the commit history.
- It allows you to incorporate changes from one branch onto another while maintaining a cleaner and more linear commit history.

Why Use Git Rebase?

- Git rebase is commonly used to integrate feature branches into the main branch (e.g., master) by replaying commits on top of the latest changes.
- It helps streamline the commit history, making it easier to review, understand, and maintain over time.

How Does Git Rebase Work?

- Git rebase works by taking a series of commits from one branch (the source branch) and applying them onto another branch (the target branch) one by one.
- It replays each commit individually, effectively transplanting them onto the target branch.
- Git rebase can also be used to squash, edit, or reorder commits, providing flexibility in history manipulation.

Basic Usage of Git Rebase

- To perform a basic rebase, use the command `git rebase <target_branch>` while on the source branch.
- This command replays the commits from the source branch onto the target branch, incorporating the changes seamlessly.

Interactive Rebase

- Git rebase offers an interactive mode (`git rebase -i`) that allows you to interactively reorder, edit, squash, or drop commits during the rebase process.
- This gives you granular control over the commit history and enables you to create a cleaner and more organized history.

Squashing Commits

- One common use case for interactive rebase is squashing commits, where multiple commits are combined into a single commit.
- This helps reduce commit clutter and create more meaningful and atomic commits.

Editing Commits

- Interactive rebase also allows you to edit commit messages or modify the content of individual commits.
- You can split, merge, or reorder commits as needed to improve the clarity and coherence of the commit history.

Reordering Commits

- Another useful feature of interactive rebase is the ability to reorder commits.
- This can help organize commits logically or group related changes together for better readability.

Git Cherry-pick

Introduction to Git Cherry-Pick

- Git cherry-pick is a command in Git that allows you to apply a specific commit from one branch onto another branch.
- It enables you to select individual commits and apply them to a different branch, allowing for the selective integration of changes.

Why Use Git Cherry-Pick?

- Git cherry-pick is useful when you want to selectively apply changes from one branch to another without merging the entire branch.
- It's commonly used for backporting bug fixes, applying hotfixes, or incorporating specific features into stable branches.

How Does Git Cherry-Pick Work?

- Git cherry-pick works by copying the changes introduced by a specific commit and applying them as a new commit onto the current branch.
- It creates a new commit with the same changes as the original commit, preserving the commit message and authorship.

Basic Usage of Git Cherry-Pick

- To cherry-pick a commit, use the command `git cherry-pick <commit_hash>`.
- This command applies the changes introduced by the specified commit onto the current branch, creating a new commit.

Multiple Cherry-Picks

- Git cherry-pick also supports cherry-picking multiple commits in a single operation.
- Simply specify the commit hashes of the desired commits separated by spaces, e.g., `git cherry-pick <commit1> <commit2> <commit3>`.

Resolving Conflicts

- During a cherry-pick operation, conflicts may arise if the changes in the cherry-picked commit conflict with the current state of the branch.
- Git provides tools to resolve these conflicts manually, allowing you to ensure the integrity of the codebase before completing the cherry-pick.

Picking Commits Across Branches

- Git cherry-pick allows you to pick commits from any branch, not just the current branch.
- You can specify the branch name along with the commit hash to cherry-pick commits from a different branch, e.g., `git cherry-pick <branch_name>^..<branch_name>`.