# GIT CHEAT SHEET

FLAVIO COPES

For inquiries, please contact: Flavio Copes flavio@flaviocopes.com

# Preface

Welcome to the Git Cheat Sheet, an extensive guide crafted to empower both novice and seasoned developers with the knowledge needed to effectively utilize Git, the most popular version control system in the software industry. This cheat sheet is designed to be your go-to resource, whether you're managing a solo project or collaborating within a large team. By providing clear explanations and practical examples, it aims to demystify Git's complexities and transform them into intuitive, actionable insights.

Throughout this guide, you will explore a wide array of Git commands and concepts that form the backbone of software version control. From fundamental operations like initializing repositories and committing changes, to more advanced techniques such as branching, merging, and rebasing, this cheat sheet covers it all. You'll also delve into specialized topics like squashing commits, bisecting to debug, handling submodules, and implementing subtrees, ensuring you're well-prepared to tackle any challenge that arises in your development process.

As you progress, you'll learn how to maintain data integrity, manage multiple working trees, and resolve merge conflicts efficiently. Each section is structured to provide step-by-step guidance, empowering you to apply what you learn immediately in your projects. By the end of this journey, you'll not only have a deeper understanding of Git but also the confidence to use it to streamline your workflow and enhance collaboration with peers.

To fully benefit from this cheat sheet, it is recommended that readers possess a foundational knowledge of command-line operations and general programming principles. Familiarity with using a terminal or command prompt will significantly aid in understanding and applying the examples provided. Additionally, having a basic grasp of version control concepts will enhance your ability to navigate through this guide effectively.

# Basic Git Commands

In this section, we delve into the fundamental Git commands that serve as the building blocks for efficiently managing and navigating your Git repositories. Git, a distributed version control system, is essential for tracking changes in your codebase, collaborating with other developers, and maintaining the integrity of your project history. Understanding these basic commands is

crucial for anyone looking to leverage the full power of Git in their development workflow.

We'll explore a variety of commands that cover key aspects of Git usage, such as initializing new repositories, committing changes, branching, and merging. Each command is explained with a short sentence that describes its purpose and practical examples to illustrate how it can be effectively used in real-world scenarios. Whether you're setting up a new project or working on an existing codebase, these commands will help you keep your work organized and maintain a seamless workflow.

## `git help`

The `git help` command prints the Git help information. It provides a quick reference to Git's basic usage and the most commonly used Git commands. This command is useful when you need a quick reminder of Git's functionality or want to explore available commands.

You can also use `git help <command>` to display help information for any specific Git command. For example, `git help git` prints the Git help specifically for the Git command itself.

These help commands are valuable resources for both beginners and experienced users to quickly access information about Git's features and usage.

## `git version`

The `git version` command displays the version of Git installed on your system. This command is useful for verifying which version of Git you are currently using, which can be important for compatibility with certain features or when troubleshooting issues.

## `git init`

The command `git init` is used to initialize a new Git repository in the current directory. This command creates a new subdirectory named `.git` that contains all the necessary metadata for the new repository. It's typically the first command you run when starting a new project that you want to manage with Git. After running this command, you can begin tracking files and making commits in your new Git repository.

## `git clone <repository_url>`

The `git clone <repository_url>` command creates a copy of a remote Git repository on your local machine. It downloads all the files, branches, and commit history, allowing you to start working with the project immediately.

### `git status`

The `git status` command shows the current state of the Git repository's working directory and staging area. It displays information about which files have been modified, added, or deleted, and whether these changes are staged for the next commit.

## The Working Directory and the Staging Area

The working directory and the staging area are fundamental concepts in Git that play crucial roles in the version control process. The working directory is the environment where you actively make changes to your files, representing the current state of your project. It is essentially a sandbox where you can freely edit, delete, and create files as you develop your project. However, these changes are local to your machine and not yet part of the version history.

On the other hand, the staging area, also known as the index, serves as an intermediary space between the working directory and the repository. It acts as a checkpoint where you can selectively organize changes before they are committed to the repository's history. This allows you to prepare a set of changes that are logically related, ensuring that each commit is meaningful and coherent.

The commands below will facilitate the management of changes between the working directory and the staging area. They enable you to add files to the staging area, remove them, or modify the existing ones, giving you control over what will be included in the next commit. By using these commands, you can ensure that only the intended updates are committed, making your project's history clear and organized. This process is essential for maintaining a clean and understandable history, as it allows you to track the evolution of your project with precision and clarity.

### `git checkout .`

The `git checkout .` command discards all changes in the working directory, reverting files to their last committed state.This command is useful for quickly

undoing local modifications and restoring the working directory to a clean state.

### `git reset -p`

The `git reset -p` command allows you to interactively reset changes in the working directory. It provides a way to selectively undo modifications, giving you fine-grained control over which changes to keep or discard.

### `git add <file>`

The `git add <file>` command adds a specific file to the staging area in Git. This prepares the file for inclusion in the next commit, allowing you to selectively choose which changes to include in your version history.

### `git add -p`

Allows you to interactively stage changes from your working directory by breaking them into chunks (hunks), enabling you to review and selectively add parts of the changes to the index before committing.

### `git add -i`

Enters the interactive mode of adding files. Provides a text-based interactive menu where you can select various actions to perform, such as staging individual changes, updating files, or viewing the status.

### `git rm <file>`

Remove a file from the working directory and stages the removal

### `git rm --cached <file>`

Removes the specified file from the staging area (index) but leaves it intact in your working directory, effectively untracking the file from version control.

### `git mv <old_path> <new_path>`

The `git mv <old_path> <new_path>` command is used to move or rename a file or directory within a Git repository. It automatically stages the change, making it ready for the next commit.

### `git commit -m "message"`

The `git commit -m "message"` command is used to create a new commit in your Git repository. It saves the changes that have been staged (added to the index)

along with a descriptive message. This message should briefly explain what changes were made in this commit.

# Working with Branches

Git branches are parallel lines of development within a Git repository, allowing you to work on different features, bug fixes, or experiments independently from the main codebase. Each branch can have its own commit history, and changes made in one branch do not affect others until they are merged. This helps in organizing work and facilitates collaboration by enabling multiple developers to work on different aspects of a project simultaneously without interfering with each other's progress.

In this section we'll introduce commands that allow you to create, switch, list, rename, and delete branches in your Git repository. These commands help manage parallel lines of development, enabling you to work on different features or bug fixes independently. You'll also learn how to display commit histories and branch relationships, as well as manage remote branches.

`git branch <branch_name>`

Create a new branch

`git checkout <branch_name>`

Switch to the specified branch and update the working directory

`git branch`

List all branches

`git branch -d <branch_name>`

Delete a branch

`git push --delete <remote> <branch>`

Delete a remote branch

`git branch -m <old_name> <new_name>`

Rename a branch

`git checkout -b <new_branch>`

Create and switch to a new branch named <new_branch>, based on the current branch.

`git switch <branch>`

Switches the working directory to the specified branch

`git show-branch <branch>`

Display a summary of the commit history and branch relationships for all or selected branches, showing where each branch diverged

`git show-branch --all`

Same as above, but for all branches and their commits.

`git branch -r`

List all remote branches that your local repository is aware of

`git branch -a`

Lists all branches in your repository, including both local and remote branches (the ones the local repository is aware of)

`git branch --merged`

Lists all branches that have been fully merged into the current branch, and can be safely deleted if no longer needed

`git branch --no-merged`

List all branches that have not been fully merged into your current branch, showing branches with changes that haven't been integrated yet

# Merging

The git merge command is used to combine the changes from one branch into another branch. It integrates the histories of both branches, creating a new commit that includes the changes from both sources. This process allows multiple lines of development to be brought together, facilitating collaboration and ensuring that all updates are incorporated into the main project. During a merge, conflicts may arise if changes overlap, requiring manual resolution to ensure a coherent final result.

`git merge <branch>`

Integrate the changes from the specified branch into your current branch, combining their histories

`git merge --no-ff <branch>`

Merge the specified branch into your current branch, always creating a new merge commit even if a fast-forward merge is possible

`git merge --squash <branch>`

Combines all the changes from the specified branch into a single commit, preparing the changes for commit in the current branch without merging the branch's history, allowing you to manually edit the commit message.

`git merge --abort`

Cancels an ongoing merge process and restores the state of your working directory and index to what it was before the merge started.

`git merge -s ours <branch>`

`git merge —-strategy=ours <branch>`

Performs a merge using the "ours" strategy, which keeps the current branch's changes and discards changes from the specified branch, effectively merging the histories without integrating the changes from the other branch.

`git merge --strategy=theirs <branch>`

Merges the specified branch into the current branch using the "theirs" strategy, which resolves all conflicts by favoring changes from the branch being merged (note: "theirs" strategy is not a built-in strategy and usually requires custom scripting or is used with tools to handle conflict resolution).

# Remotes

Git remotes are references to remote repositories, which are versions of your project hosted on the internet or another network. They enable collaboration by allowing multiple users to share and sync changes with a central repository. Common operations with remotes include git fetch to retrieve updates, git pull to fetch and merge changes, and git push to upload local commits to the

remote repository. Managing remotes involves adding, removing, and renaming remote connections, as well as configuring URLs for seamless collaboration.

`git fetch`

Fetch changes from a remote repository but do not merge them into your current branch

`git pull`

Fetch changes from a remote repository and immediately merges them into your current branch

`git push`

Upload your local branch's changes to a remote repository

`git remote`

List the names of remote repositories configured for your local repository

`git remote -v`

Display the URLs of the remote repositories associated with your local repository, showing both the fetch and push URLs

`git remote add <name> <url>`

Add a new remote repository with the specified name and URL to your local repository configuration.

`git remote remove <name>`

`git remote rm <name>`

Deletes the specified remote repository connection from your local git configuration

`git remote rename <old_name> <new_name>`

Changes the name of an existing remote repository connection in your local git configuration

`git remote set-url <name> <newurl>`

Changes the URL of an existing remote repository connection in your local git configuration

`git fetch <remote>`

Retrieves the latest changes from the specified remote repository, updating your local copy of the remote branches without merging them into your local branches.

`git pull <remote>`

Fetches changes from the specified remote repository and merges them into your current branch

`git remote update`

Fetches updates for all remotes tracked by the repository.

`git push <remote> <branch>`

Uploads the specified branch from your local repository to the given remote repository

`git push <remote> --delete <branch>`

Removes the specified branch from the remote repository

`git remote show <remote>`

Displays detailed information about the specified remote repository, including its URL, fetch and push configurations, and the branches it tracks

`git ls-remote <repository>`

Lists references (such as branches and tags) and their commit IDs from the specified remote repository. This command allows you to view the branches and tags available in a remote repository without cloning it.

`git push origin <branch> --set-upstream`

Pushes the local branch <branch> to the remote repository origin and sets up the local branch to track the remote branch, so future git push and git pull commands will default to this remote branch.

`git remote add upstream <repository>`

Adds a new remote named upstream to your local repository, pointing to the specified <repository>. This is commonly used to track the original repository from which you forked, while origin typically refers to your own fork.

`git fetch upstream`

Retrieves updates from the upstream remote repository, updating your local references to the branches and tags from that remote without modifying your working directory or merging changes.

`git pull upstream <branch>`

Fetches updates from the <branch> of the upstream remote repository and merges those changes into your current branch. This is often used to integrate changes from the original repository into your own local branch.

`git push origin <branch>`

Uploads the local branch <branch> to the origin remote repository, making your branch and its commits available on the remote.

# Amending Commits

Amending Git commits allows you to modify the most recent commit, typically to correct or update its contents or message. This can be done using the git commit --amend command, which opens the commit in your default text editor for changes. Amending is particularly useful for fixing small mistakes or adding forgotten changes without creating a new commit, resulting in a cleaner and more accurate commit history.

`git commit --amend`

Modify the most recent commit, combining staged changes.

`git commit --amend -m "new message"`

Amends the commit message of the most recent commit.

`git commit --fixup=HEAD`

Creates a new commit with the `--fixup` option that is intended to correct or amend the most recent commit (HEAD). The new commit is marked with a fixup! prefix in the commit message and will be used to automatically fix or amend the specified commit during an interactive rebase.

# Stashing

Git stashing is a feature that allows you to temporarily save changes in your working directory that are not yet ready to be committed. By using the git stash command, you can set aside these changes and revert your working directory to a clean state, enabling you to switch branches or perform other tasks without losing progress. Later, you can reapply the stashed changes with git stash apply or git stash pop, allowing you to continue where you left off. This functionality is especially useful for managing work in progress when you need to address an urgent issue or experiment with a different code path.

`git stash`

`git stash save`

Temporarily save your uncommitted changes, allowing you to switch branches or perform other operations without committing incomplete work.

`git stash -m "message"`

`git stash save "message"`

Same as above, but stores changes with a message

`git stash show`

Display a summary of the changes in the most recent stash entry, showing which files were modified

`git stash list`

Show all the stashed changes in your repository, displaying them in a numbered list

`git stash pop`

Apply the most recent stash and then immediately remove it from the stash list

`git stash drop`

Remove the most recent stash entry from the stash list without applying it to your working directory

`git stash apply`

Reapply the most recently stashed changes to your working directory without removing them from the stash list

`git stash clear`

Clear remove all stashed entries, permanently deleting all saved changes in your stash l

`git stash branch <branch>`

Creates a new branch named <branch> from the commit where you were before stashing your changes, and then it applies the stashed changes to that new branch. This command effectively allows you to continue working on your stashed changes in a separate branch, preserving the original context and changes.

# Tagging

Git tagging is a feature that allows you to mark specific points in your repository's history as important with a meaningful name, often used for releases or significant milestones. Unlike branches, tags are typically immutable and do not change, serving as a permanent reference to a particular commit. There are two types of tags in Git: lightweight tags, which are simple pointers to a commit, and annotated tags, which store additional metadata like the tagger's name, email, date, and a message. Tags can be easily created, listed, pushed to remote repositories, and deleted, providing a convenient way to manage and reference key points in your project's development timeline.

`git tag <tag_name>`

Create a new tag with the specified name pointing to the current commit (typically used to mark specific points in the commit history, like releases)

`git tag -a <tag_name> -m "message"`

Create an annotated tag with the specified name and message, which includes additional metadata like the tagger's name, email, and date, and points to the current commit.

`git tag -d <tag_name>`

Deletes the specified tag from your local repository

`git tag -f <tag> <commit>`

Forces a tag to point to a different commit.

`git show <tag_name>`

Display detailed information about the specified tag, including the commit it points to and any associated tag messages or annotations

`git push origin <tag_name>`

Upload the specified tag to the remote repository, making it available to others

`git push origin --tags`

Push all local tags to the remote repository, ensuring that all tags are synchronized with the remote

`git push --follow-tags`

Pushes both commits and tags.

`git fetch --tags`

Retrieve all tags from the default remote repository and updates your local repository with them, without affecting your current branches.

# Reverting Changes

Reverting changes in Git involves undoing modifications made to a repository's history. This can be accomplished through several commands, such as git revert, which creates a new commit that negates the changes of a specified previous commit, effectively reversing its effects while preserving the commit history. Another method is using git reset, which changes the current HEAD to a specified commit and can update the staging area and working directory depending on the chosen options (--soft, --mixed, or --hard). Additionally, git checkout can be used to discard changes in the working directory, reverting files to their state in the last commit. These tools provide flexibility in managing and correcting changes, ensuring the repository remains accurate and clean.

`git checkout -- <file>`

Discards changes in the specified file from the working directory, reverting it to the state of the last commit and effectively undoing any modifications that haven't been staged.

`git revert <commit>`

Creates a new commit that undoes the changes in the specified commit, effectively reversing its effects while preserving the history.

`git revert -n <commit>`

Reverts a commit but does not commit the result.

`git reset`

Resets the current HEAD to the specified state, and optionally updates the staging area and working directory, depending on the options used (--soft, --mixed, or --hard).

`git reset --soft <commit>`

Moves HEAD to the specified commit, while keeping the index (staging area) and working directory unchanged, so all changes after the specified commit remain staged for committing. This is useful when you want to undo commits but keep the changes ready to be committed again.

`git reset --mixed <commit>`

Moves HEAD to the specified commit and updates the index (staging area) to match that commit, but leaves the working directory unchanged, so changes after the specified commit are kept but untracked.

`git reset --hard <commit>`

Moves HEAD to the specified commit and updates both the index (staging area) and working directory to match that commit, discarding all changes and untracked files after the specified commit.

# Viewing History Logs

Git history refers to the record of all changes made to a repository over time. It includes a chronological sequence of commits, each representing a snapshot of the repository at a specific point. This history allows developers to track modifications, understand the evolution of the codebase, and collaborate effectively by providing a detailed log of who made changes, when, and why. Tools like git log help navigate this history, offering insights into the development process and aiding in debugging and project management.

`git log`

Display the commits log

`git log --oneline`

Displays a summary of commits with one line each.

`git log --graph`

Shows a graphical representation of the commit history.

`git log --stat`

Displays file statistics along with the commit history.

`git log --pretty=format:"%h %s"`

Formats the log output according to the specified format.

`git log --pretty=format:"%h - %an, %ar : %s"`

Provides a more human-readable format for logs

`git log --author=<author>`

Shows commits made by the specified author.

`git log --before=<date>`

Shows commits made before the specified date.

`git log --after=<date>`

`git log --since=<date>`

Shows commits made after the specified date.

`git log --cherry-pick`

Omit commits that are equivalent between two branches.

`git log --follow <file>`

Shows commits for a file, including renames.

`git log --show-signature`

Displays GPG signature information for commits.

`git shortlog`

Summarizes git log output by author.

`git shortlog -sn`

Summarizes git log output by author, with commit counts.

`git log --simplify-by-decoration`

Shows only commits that are referenced by tags or branches.

`git log --no-merges`

Omits merge commits from the log.

`git whatchanged`

Lists commit data in a format similar to a commit log.

`git diff-tree --pretty --name-only --root <commit>`

Shows detailed information for a commit tree.

`git log --first-parent`

Only show commits of the current branch and exclude those merged from other branches.

# Diffs

Git diffs are a feature in Git that allows you to see the differences between various states in your repository. This can include differences between your working directory and the staging area, between the staging area and the last commit, or between any two commits or branches. By displaying line-by-line changes in files, diffs help you review modifications before committing, merging, or applying changes, thus ensuring accuracy and consistency in your codebase.

`git diff`

Shows the differences between various states in your repository, such as between your working directory and the index (staging area), between the index and the last commit, or between two commits. It displays line-by-line changes in files, helping you review modifications before committing or merging.

`git diff --stat`

Shows a summary of changes between your working directory and the index (staging area), helping you see what files have been modified and how many lines have been added or removed.

`git diff --stat <commit>`

View Changes Between a Commit and the Working Directory

`git diff --stat <commit1> <commit2>`

Provides a summary of changes between two commits, showing which files were altered and the extent of changes between them.

`git diff --stat <branch1> <branch2>`

Summarizes the differences between the two branches, indicating the files changed and the magnitude of changes.

`git diff --name-only <commit>`

Shows only the names of files that changed in the specified commit.

`git diff --cached`

Shows the differences between the staged changes (index) and the last commit, helping you review what will be included in the next commit

`git diff HEAD`

Shows the differences between the working directory and the latest commit (HEAD), allowing you to see what changes have been made since the last commit

`git diff <branch1> <branch2>`

Shows the differences between the tips of two branches, highlighting the changes between the commits at the end of each branch

`git difftool`

Launches a diff tool to compare changes.

`git difftool <commit1> <commit2>`

Uses the diff tool to show the differences between two specified commits.

`git difftool <branch1> <branch2>`

Opens the diff tool to compare changes between two branches.

`git cherry <branch>`

Compares the commits in your current branch against another branch and shows which commits are unique to each branch. It is commonly used to identify which commits in one branch have not been applied to another branch.

# Git Flow

Git Flow is a branching model for Git that provides a robust framework for managing larger projects. It defines a strict branching strategy designed around the project release cycle, with two primary branches (main and develop) and supporting branches for features, releases, and hotfixes. This model helps in organizing work, ensuring a clean and manageable history, and facilitating collaboration by clearly defining roles and processes for different types of development work.

`git flow init`

Initializes a repository for git-flow branching model.

`git flow feature start <feature>`

Starts a new feature branch in git-flow.

`git flow feature finish <feature>`

Finishes a feature branch in git-flow.

# Exploring Git References

Git references, often referred to as refs, are pointers to specific commits or objects within a Git repository. These can include branches, tags, and other references like HEAD, which points to the current commit checked out in your working directory. References are used to keep track of the structure and history of the repository, enabling Git to efficiently manage and navigate different points in the project's timeline. They provide a way to name and refer to particular commits, making it easier to work with and manipulate the repository's history.

`git show-ref --heads`

Lists references to all heads (branches).

`git show-ref --tags`

Lists references to all tags.

# Configuration

Git configuration involves setting up various options and preferences that control the behavior of your Git environment. This can include specifying your username and email, setting up default text editors, creating aliases for commonly used commands, and configuring global ignore files. Configuration settings can be applied at different levels: global (affecting all repositories on your system), local (affecting a single repository), and system-wide. These settings ensure a customized and consistent user experience, streamline workflows, and enhance the overall efficiency of version control operations.

`git config --global user.name "Your Name"`

Sets the user name on a global level.

`git config --global user.email "your_email@example.com"`

Sets the user email on a global level.

`git config --global core.editor <editor>`

Sets the default text editor.

`git config --global core.excludesfile <file>`

Sets the global ignore file.

`git config --list`

Lists all the configuration settings.

`git config --list --show-origin`

Lists all config variables, showing their origins.

`git config <key>`

Retrieves the value for the specified key.

`git config --get <key>`

Retrieves the value for the specified configuration key.

`git config --unset <key>`

Removes the specified configuration key.

`git config --global --unset <key>`

Removes the specified configuration key globally.

# Security

Git GPG security involves using GNU Privacy Guard (GPG) to sign commits and tags, ensuring their authenticity and integrity. By configuring a GPG key and enabling automatic signing, developers can verify that commits and tags were created by a trusted source, preventing tampering and ensuring the integrity of the repository's history. This practice enhances security by providing cryptographic assurance that the changes come from a legitimate contributor.

`git config --global user.signingKey <key>`

Configures the GPG key for signing commits and tags.

`git config --global commit.gpgSign true`

Automatically signs all commits with GPG.

# Setting Aliases

Git aliases are custom shortcuts that you can create to simplify and speed up your workflow by mapping longer Git commands to shorter, more memorable names. By configuring aliases in your Git settings, you can quickly execute frequently used commands with less typing. This not only enhances productivity but also reduces the likelihood of errors. For example, you can set an alias like git st to replace git status, or git co to replace git checkout. Aliases can be defined globally to apply across all repositories or locally for individual projects, providing flexibility in how you streamline your Git operations.

`git config --global alias.ci commit`

Sets git ci as an alias for git commit.

`git config --global alias.st status`

Sets git st as an alias for git status.

`git config --global alias.co checkout`

Sets git co as an alias for git checkout.

`git config --global alias.br branch`

Sets git br as an alias for git branch.

`git config --global alias.graph "log --graph --all --oneline --decorate"`

Creates an alias for a detailed graph of the repository history.

# Rebasing

Git rebasing re-applies your changes on top of another branch's history, creating a cleaner and more linear project history. In practice, this helps integrate updates smoothly by avoiding unnecessary merge commits, ensuring that the commit sequence is straightforward, and making it easier to understand the evolution of the project.

`git rebase <branch>`

The git rebase command is used to re-apply commits on top of another base tip. It allows you to move or combine a sequence of commits to a new base commit. This is commonly used to:

1. Keep a linear project history.
2. Integrate changes from one branch into another.
3. Update a feature branch with the latest changes from the main branch.

The basic usage is git rebase <branch>, which will rebase the current branch onto the specified branch.

`git rebase --interactive <branch>`

Starts an interactive rebase session, allowing you to modify commits starting from <base> up to the current HEAD. This lets you reorder, squash, edit, or delete commits, providing a way to clean up and refine commit history before pushing changes. Shorter version: `git rebase -i <branch>`

`git rebase --continue`

Continues the rebase process after resolving conflicts.

`git rebase --abort`

Aborts the rebase process and returns to the original branch.

`git fetch --rebase`

Fetches from the remote repository and rebases local changes.

# Cherry-Picking

Git cherry-picking is a process that allows you to apply the changes introduced by a specific commit from one branch into another branch. This is particularly useful when you want to selectively incorporate individual changes from different branches without merging the entire branch. By using the git cherry-pick command, you can isolate and integrate only the desired commits, ensuring that specific modifications are included in your current branch while avoiding potential conflicts and unwanted changes from other parts of the branch.

`git cherry-pick <commit>`

Applies the changes introduced by an existing commit.

`git cherry-pick --continue`

Continues cherry-picking after resolving conflicts.

`git cherry-pick --abort`

Aborts the cherry-pick process.

`git cherry-pick --no-commit <commit>`

Cherry-picks a commit without automatically committing and allows further changes. Shorter version: `git cherry-pick -n <commit>`

# Patching

Git patching is a method used to apply changes from one repository to another or from one branch to another within the same repository. It involves creating patch files, which are text files representing differences between commits or

branches. These patch files can then be applied to a repository using commands like git apply or git am, allowing changes to be transferred and integrated without directly merging branches. Patching is particularly useful for sharing specific changes or updates across different codebases, ensuring that only the intended modifications are applied.

`git apply <patch_file>`

Applies changes to the working directory from a patch file.

`git apply --check`

Checks if patches can be applied cleanly.

`git format-patch <since_commit>`

Creates patch files for each commit since the specified commit.

`git am <patch_file>`

Applies patches from a mailbox.

`git am --continue`

Continues applying patches after resolving conflicts.

`git am --abort`

Aborts the patch application process.

`git diff > <file.patch>`

Creates a patch file from differences.

# Relative dates

Git relative dates allow users to refer to specific points in the repository's history using human-readable time expressions. For instance, commands like main@{1.week.ago} or @{3.days.ago} enable you to access the state of a branch or view changes made since a certain time period relative to the current date. This feature simplifies navigating the repository's timeline by using intuitive terms like "yesterday," "2 weeks ago," or specific dates, making it easier to track and manage the evolution of the codebase without needing to remember exact commit hashes or timestamps.

`git show main@{1.week.ago}`

See the state of your main branch one week ago:

`git diff @{3.days.ago}`

See what changes you've made in the last 3 days:

`git checkout main@{2.weeks.ago}`

Check out your repository as it was 2 weeks ago:

`git log @{1.month.ago}..HEAD`

See the log of commits from 1 month ago until now

`@{2024-06-01}`

`@{yesterday}`

`@{"1 week 2 days ago"}`

Other usage examples.

# Blaming

Git blaming is a feature in Git that identifies the last modification made to each line of a file, attributing changes to specific commits and authors. This is done using the git blame command, which provides a detailed annotation of the file, showing who made changes and when they were made. This tool is particularly useful for tracking the history of a file, understanding the evolution of the code, and identifying the source of bugs or changes. By pinpointing the exact commit and author responsible for each line, developers can gain insights into the development process and facilitate better collaboration and accountability within a team.

`git blame <file>`

Shows the last modification for each line of a file.

`git blame <file> -L <start>,<end>`

Limits the blame output to the specified line range.

`git blame <file> <commit>`

Shows the blame information up to the specified commit.

`git blame <file> -C -C`

Shows which revisions and authors last modified each line of a file, with copying detection.

The `-C` option detects lines moved or copied within the same file. Using it once (`-C`) detects lines moved or copied within the same file. Using the `-C` option twice (`-C -C`) makes git inspect unmodified files as candidates for the source of copy. This means it will try to find the origin of copied lines not just in the same file but in other files as well.

`git blame <file> --reverse`

Works backwards, showing who last altered each line in the specified file.

`git blame <file> --first-parent`

Shows who most recently modified each line in a file, following only the first parent commit for merge changes.

# Archiving

Git archiving is a feature that allows you to create archive files, such as .tar or .zip, containing the contents of a specific commit, branch, or tag. This is useful for packaging a snapshot of your repository at a specific point in time, enabling you to distribute or backup the repository's state without including the entire Git history. The git archive command is typically used for this purpose, providing a convenient way to export the current state of the project into a portable format.

`git archive <format> <tree-ish>`

creates an archive file (e.g., a .tar or .zip file) containing the contents of the specified tree-ish (like a commit, branch, or tag) in the given format. For example:

`git archive --format=tar HEAD` creates a .tar archive of the current commit (HEAD).

`git archive --format=zip v1.0` creates a .zip archive of the files in the v1.0 tag.

This command is useful for packaging a snapshot of your repository at a specific point in time.

# Tracking

Git tracking refers to the process of monitoring and managing the files in a repository. The command git ls-files lists all files that are being tracked by Git, providing a clear view of the files that are currently under version control. On the other hand, git ls-tree <branch> displays the contents of a tree object for a specified branch, showing the structure and files at that point in the repository. Together, these commands help developers understand which files are included in the repository and how they are organized, ensuring efficient tracking and management of the project's codebase.

`git ls-files`

Lists all tracked files.

`git ls-tree <branch>`

Lists the contents of a tree object.

# Index Manipulation

Git index manipulation involves managing the staging area (also known as the index) where changes are prepared before committing. This can include marking files as "assume unchanged" to temporarily ignore changes, or resetting these markings to track changes again. Index manipulation commands, such as git update-index, allow you to control which files are included in the next commit, providing flexibility in handling changes and optimizing the workflow for specific tasks.

`git update-index --assume-unchanged <file>`

Marks a file as assume unchanged.

`git update-index --no-assume-unchanged <file>`

Unmarks a file as assume unchanged.

# Squashing

Git squashing is the process of combining multiple commits into a single commit. This is often done to clean up the commit history before merging changes into a main branch, making the history more concise and easier to read. Squashing can be performed using the interactive rebase command (git rebase -i), which allows developers to selectively merge, reorder, or edit

commits. By squashing commits, redundant or minor changes can be consolidated, presenting a clearer narrative of the development process.

`git rebase -i HEAD~<n>`

Squashes commits interactively.

# Data Integrity

Git data integrity refers to the mechanisms and processes Git employs to ensure the accuracy and consistency of data within a repository. Git uses cryptographic hashes (SHA-1 or SHA-256) to uniquely identify objects such as commits, trees, and blobs. This hashing not only provides a unique identifier for each object but also ensures that any modification to the object's content will result in a different hash, thus detecting any corruption or tampering. Commands like git fsck can be used to verify the connectivity and validity of objects in the database, ensuring the overall health and integrity of the repository.

`git fsck`

Verifies the connectivity and validity of objects in the database.

`git fsck --unreachable`

Finds objects in the repository that are not reachable from any reference.

`git prune`

Remove unreachable objects

`git gc`

Runs a garbage collection process.

Git garbage collection is a maintenance process that cleans up and optimizes the repository by removing unnecessary files and compressing file revisions to save space. This process, triggered by the git gc command, consolidates and deletes unreachable objects, such as orphaned commits and unreferenced blobs, ensuring the repository remains efficient and performant. Regular garbage collection helps manage storage effectively and keeps the repository's structure organized.

# Cleanup

Cleaning up in Git involves removing unnecessary files, references, and branches that are no longer needed. This helps to keep your repository organized and efficient. Regular cleanup activities, such as pruning remote-tracking branches, deleting untracked files, and removing stale references, ensure that your repository remains manageable and free from clutter. In practice, these actions can improve performance, reduce storage requirements, and make it easier to navigate and work within your project.

`git fetch --prune`

Removes references that no longer exist on the remote.

`git remote prune <name>`

Prunes all stale remote-tracking branches.

`git fetch origin --prune`

Cleans up outdated references from the remote repository.

`git clean -f`

Removes untracked files from the working directory, forcing the deletion of files not being tracked by Git.

`git clean -fd`

Removes untracked files and directories from the working directory, including any files and directories not tracked by Git.

`git clean -i`

Enters interactive mode for cleaning untracked files.

`git clean -X`

Removes only ignored files from the working directory.

# Subtree

Git subtree is a mechanism for managing and integrating subprojects into a main repository. Unlike submodules, which treat the subproject as a separate entity with its own repository, subtrees allow you to include the contents of

another repository directly within a subdirectory of your main repository. This approach simplifies the workflow by eliminating the need for multiple repositories and enabling seamless integration, merging, and pulling of updates from the subproject. Subtrees provide a flexible and convenient way to manage dependencies and collaborate on projects that require incorporating external codebases.

`git subtree add --prefix=<dir> <repository> <branch>`

Adds a repository as a subtree.

`git subtree merge --prefix=<dir> <branch>`

Merges a subtree.

`git subtree pull --prefix=<dir> <repository> <branch>`

Pulls in new changes from the subtree's repository.

# Searching

git grep is a powerful search command in Git that allows users to search for specific strings or patterns within the files of a repository. It searches through the working directory and the index, providing a quick and efficient way to locate occurrences of a specified pattern across multiple files. This command is particularly useful for developers looking to find instances of code, comments, or text within a project, enabling them to navigate and understand large codebases with ease. With various options and flags, git grep can perform targeted searches, making it an essential tool for code analysis and maintenance.

`git grep <pattern>`

Searches for a string in the working directory and the index.

`git grep -e <pattern>`

Searches for a specific pattern.

# Bisecting

Git bisecting is a powerful debugging tool that helps identify the specific commit that introduced a bug or issue in a project. By performing a binary

search through the commit history, git bisect efficiently narrows down the range of potential problem commits. The process involves marking a known good commit and a known bad commit, and then repeatedly testing intermediate commits to determine whether they are good or bad. This iterative approach quickly isolates the faulty commit, allowing developers to pinpoint the exact change that caused the problem, thereby facilitating faster and more accurate debugging.

`git bisect start`

Starts a bisect session.

`git bisect bad`

Marks the current version as bad.

`git bisect good <commit>`

Marks the specified commit as good.

`git bisect reset`

Ends a bisect session and returns to the original branch.

`git bisect visualize`

Launches a visual tool to assist with bisecting.

# Attributes

Git attributes are settings that define how Git should handle specific files or paths within a repository. These attributes are defined in a file named .gitattributes, and they can control various behaviors such as text encoding, line-ending normalization, merge strategies, and diff algorithms. By setting attributes, you can ensure consistent behavior across different environments and collaborators, making it easier to manage files with special requirements or complexities. For example, you can mark certain files as binary to prevent Git from attempting to merge them, or specify custom diff drivers for more meaningful comparisons.

`git check-attr <attribute> -- <file>`

Shows the value of a specific attribute for the given file as defined in the .gitattributes configuration, helping you understand how Git is treating the file

with respect to attributes like text encoding, merge behavior, or diff handling.

# Checkout

git checkout is a versatile command in Git used to switch between different branches, tags, or commits within a repository. By updating the working directory and index to match the specified branch or commit, it allows you to view or work with the state of the repository at that point. Additionally, git checkout can be used to create new branches, restore specific files from a commit, or even start a new branch with no history using the --orphan option. This command is essential for navigating and managing different versions of a project's codebase.

`git checkout <commit>`

Updates the working directory and index to match the specified commit, allowing you to view or work with the state of the repository at that commit, but it leaves you in a "detached HEAD" state, meaning you're not on any branch

`git checkout -b <branch> <commit>`

Creates a new branch named <branch> starting from the specified commit and switches to that branch, allowing you to begin working from that point in the commit history.

`git checkout <commit> -- <file>`

Restores the specified file from a specific commit into your working directory, replacing the current version of the file with the version from that commit without changing the commit history or index.

`git checkout --orphan <new_branch>`

Creates a new branch named <new_branch> with no commit history, effectively starting a new branch that begins with a clean working directory and index, as if it were a new repository.

# Reflog

Git reflog is a powerful tool that records all changes made to the tips of branches and the HEAD reference in a Git repository. This includes actions such as commits, checkouts, merges, and resets. By maintaining a history of

these changes, reflog allows users to track recent modifications and recover lost commits, even if they are not part of the current branch history. It provides a way to navigate through the repository's state changes, making it an invaluable resource for debugging and undoing mistakes.

`git reflog`

Displays a log of all the changes to the HEAD reference and branch tips, including commits, checkouts, merges, and resets, allowing you to recover lost commits or track recent changes to the repository's state

`git reflog show <ref>`

Displays the reflog for the specified reference (<ref>), showing a log of changes to that reference, including updates to HEAD or branch tips, along with associated commit messages and timestamps.

# Handling Untracked Files

`git clean`

Remove untracked files and directories from the working directory. By default, it only shows what would be removed without actually deleting anything. To perform the actual cleanup, you need to use additional flags:

- `git clean -f`: Removes untracked files.
- `git clean -fd`: Removes untracked files and directories.
- `git clean -fx`: Removes untracked files, including those ignored by .gitignore.
- `git clean -n`: Shows which files would be removed without actually deleting them.

# Force Pushing

`git push --force`

Forces a push of your local branch to the remote repository, even if it results in a non-fast-forward merge. This overwrites the remote branch with your local changes. This can be necessary when you have rewritten history (e.g., with a rebase) and need to update the remote branch to match your local branch, but

it can also potentially overwrite others' changes, so it should be used with caution.

# Fetching and Pulling

`git fetch --all`

Retrieves updates from all remote repositories configured for your local repository, fetching changes from all branches and tags without modifying your local branches

`git pull --rebase`

Fetches changes from the remote repository and rebases your local commits on top of the updated remote branch, rather than merging them. This keeps the commit history linear and avoids unnecessary merge commits.

# Handling Merge Conflicts

Handling merge conflicts in Git is an essential skill for collaborating on projects with multiple contributors. Merge conflicts occur when changes in different branches or commits overlap or contradict each other, preventing an automatic merge. Resolving these conflicts involves reviewing and manually reconciling the differences to ensure that the final code integrates contributions from all parties accurately. In practice, effectively managing merge conflicts helps maintain code integrity and facilitates smooth collaboration by ensuring that everyone's changes are correctly incorporated into the project's history.

`git mergetool`

launches a merge tool to help you resolve conflicts that arise during a merge or rebase. It opens a graphical interface or a text-based tool configured in your Git settings, allowing you to manually resolve conflicts and finalize the merge.

`git rerere`

rerere stands for "reuse recorded resolution" and is a feature that helps automatically resolve conflicts in future merges or rebases by reusing conflict resolutions you've previously recorded. Once enabled, Git records how you resolved conflicts, and if the same conflicts arise again, it can apply the same resolutions automatically.

# Working trees

Working trees in Git allow you to have multiple working directories associated with a single repository. This is particularly useful for working on multiple branches simultaneously without the need to constantly switch branches in the same directory. By using working trees, you can easily manage different features, bug fixes, or experiments in isolated environments, improving workflow efficiency and reducing the risk of conflicts.

`git worktree add ../new-branch feature-branch`

Creates a new working tree in a directory named "new-branch" based on the "feature-branch".

`git worktree list`

Lists all working trees associated with the current repository, showing their paths and the branches they are checked out to.

`git worktree remove <path>`

Removes the specified working tree at the given <path>, deleting the working directory and detaching the branch.

`git worktree prune`

Removes references to nonexistent working trees, cleaning up the working tree list.

`git worktree lock <path>`

Locks the specified working tree at the given <path>, preventing it from being pruned.

`git worktree unlock <path>`

Unlocks the specified working tree at the given <path>, allowing it to be pruned if necessary.

# Submodules

Submodules in Git are a way to include and manage external repositories within your own repository. They are particularly useful for reusing code across multiple projects, maintaining dependencies, or integrating third-party libraries.

By using submodules, you can keep your main repository clean and modular, while still ensuring that all necessary components are included and version-controlled.

`git submodule init`

Initializes submodules in your repository. This command sets up the configuration necessary for the submodules, but doesn't actually clone them.

`git submodule update`

Clones and checks out the submodules into the specified paths. This is typically run after git submodule init.

`git submodule add <repository> <path>`

Adds a new submodule to your repository at the specified path, linking it to the specified repository.

`git submodule status`

Displays the status of all submodules, showing their commit hashes and whether they are up-to-date, modified, or uninitialized.

`git submodule foreach <command>`

Runs the specified command in each submodule. This is useful for performing batch operations across all submodules.

`git submodule sync`

Synchronizes the submodule URLs in your configuration file with those in the .gitmodules file, ensuring they are up-to-date.

`git submodule deinit <path>`

Unregisters the specified submodule, removing its configuration. This doesn't delete the submodule's working directory.

`git submodule update --remote`

Fetches and updates the submodules to the latest commit from their remote repositories.

`git submodule set-url <path> <newurl>`

Changes the URL of the specified submodule to the new URL.

`git submodule absorbgitdirs`

Absorbs the submodule's Git directory into the superproject to simplify the structure.